DEPARTAMENTO DE INGENIERÍA TELEMÁTICA Y ELECTRÓNICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE
TELECOMUNICACIÓN

Automatic Runtime Performance Optimization Methodology Applied to Dataflow-based Hyperspectral Imaging Cancer Detection Algorithms

TESIS DOCTORAL

Autor: Raquel Lazcano López

Graduada en Ingeniería Electrónica de Comunicaciones por la Universidad Politécnica de Madrid
Máster Universitario en Ingeniería de Sistemas y Servicios para la Sociedad de la Información por la
Universidad Politécnica de Madrid

Director: Eduardo Juárez Martínez

Docteur Ès Sciences Techniques por l'École Polytechnique Fédéral de Lausanne

Director: César Sanz Álvaro

Doctor Ingeniero de Telecomunicación por la Universidad Politécnica de Madrid

2020

Tribunal

Tribunal nombrado por el Mgfco. y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, el día de de .

Presidente: –

Vocales: –

–

–

Secretario: –

Suplentes: –

–

Realizado el acto de lectura y defensa de la Tesis el día de de en la Escuela Técnica Superior de Ingeniería y Sistemas de Telecomunicación de la Universidad Politécnica de Madrid.

Calificación:

EL PRESIDENTE

LOS VOCALES

EL SECRETARIO

A mis padres, por darlo todo y serlo todo, siempre

A mi abuela, por todos los «si mi padre te viera...»

*Y a mi abuelo, porque, aunque no puedas decirlo,
sé que tú también estás orgulloso de mí*

Agradecimientos

Durante todos estos años, me he imaginado muchas veces este momento, sentada frente a mi ordenador, intentando encontrar las palabras que expresen cómo me siento. Todavía me cuesta creer que ya ha llegado, que con esto cerraré la etapa más larga de mi vida, en la que he crecido como profesional, pero sobre todo como persona. Hacer una tesis es un proceso largo, arduo, apasionante y frustrante a la vez, que te hace querer tirar la toalla muchas veces. Si no lo he hecho ha sido gracias a las personas que me han acompañado estos años, y por eso quiero comenzar este punto y final agradeciendo a todas ellas por recorrer este duro camino conmigo.

En primer lugar, me gustaría agradecer a mis directores de tesis, Eduardo Juárez y César Sanz. Me abristeis la puerta del laboratorio hace más de siete años, cuando aún no había terminado la carrera. Gracias también por enseñarme a no rendirme nunca, a seguir luchando a pesar de los problemas que hayan podido aparecer en el camino.

Este agradecimiento se extiende también al resto del grupo de investigación: Fer, Matías, Ángel, Pedro, Juan, gracias por haberme acogido en el grupo y haberme hecho sentir como una más desde el primer momento. Especialmente, Fer, gracias por tu cercanía y tus ánimos, sobre todo este último año (¡te quedas sin *chocolate supplier*!).

Aunque ellos también son (o han sido) parte del grupo, me gustaría agradecer de una forma muy especial a Rubén y Miguel. He tenido el placer de compartir mi día a día con vosotros desde hace ya muchos años, cuando estábamos prácticamente solos en el laboratorio. Gracias por el apoyo, el cariño, los consejos y los buenos ratos que hemos compartido. Habéis sido mi guía en el doctorado, a quien recurría siempre que tenía una duda, una queja o ganas de hablar. Gracias infinitas.

Estos años he visto crecer el laboratorio, pasando de estar casi solos a tener que hacer hueco a toda la gente nueva que ha ido llegando. Jaime, chiquitín, fuiste el primero en hacer que Dani y yo dejásemos de ser «los niños», y te has convertido en un gran amigo. Sigue como eres, siempre. Ahora, ¡te paso el testigo! Gema, Guille, Luisa, Marta, Anup, Gonzalo... gracias por tantos buenos ratos. Os deseo lo mejor.

Una de las ventajas de la tesis es que te permite colaborar con muchas personas de otras instituciones, y me gustaría agradecer a todas ellas. Guardo especial cariño a Himar, Samuel y Gustavo, porque fueron las primeras personas con las que colaboré, y porque con ellos aprendí lo fascinante que puede ser la investigación. Quiero agradecer también a las personas de CEI-UPM, INSA, UniCA y UniSS por la estrecha colaboración y los buenos momentos compartidos.

Durante el doctorado, he tenido la oportunidad de realizar una estancia en el extranjero, así que permitidme que cambie al inglés para agradecer esta experiencia.

During the PhD, I have had the opportunity of doing an international research stay at iCube laboratory, in Strasbourg, and I would like to thank all the people that welcomed me there. I am especially grateful to Philippe Clauss, who was my advisor. Philippe, thank you very much for all those long discussions and for everything you have taught me. I would also like to thank the rest of colleagues there: Jens, Vincent, Cédric, Arthur, Bérenger, Maxime, Luke, Harenome, thank you for everything.

My stay there allowed me not only to push my research, but also to make some very good friends. Salwa, Marek, I cannot thank you enough for all those moments at our beloved B229, all the laughs, all the talks... even all the language lessons! You have made my stay a unique experience. And June, thank you very much for all your support, your friendship... and for all the dinners we've shared in our food-tour!

Siempre he dicho que la mayor ventaja de hacer una tesis es que te permite viajar y conocer gente nueva, y creo que no puedo tener mejor ejemplo de ello. Alf, Titti, qué bonita la casualidad que nos hizo trabajar juntos. Alf, gracias por aconsejarme tanto y tan bien, gracias por haberme hecho un huequito en tu vida y gracias por tus *podcasts* infinitos. Titti, gracias por estar siempre dispuesta a escuchar, gracias por saber aconsejar y aceptar consejos (lo que es aún más difícil, y por ello te admiro más) y gracias por estar tan cerca que apenas noto que vives en otro país. Gracias a los dos, porque os habéis convertido no sólo en amigos, sino en personas imprescindibles en mi vida. Nunca podré agradecer lo suficiente el apoyo que habéis supuesto para mí estos últimos años, las veces que me habéis hecho ver las cosas de otra manera o me habéis sacado una sonrisa. Gracias por escucharme, por apoyarme y por entenderme. Sólo por poder llamaros amigos ha merecido la pena esta experiencia.

Y, cómo no, no puedo terminar estos agradecimientos sin hablar de todas las personas que llevan ya muchos años acompañándome. Marcos y Carlos, gracias por seguir al pie del cañón desde aquel primer día de carrera. Estefanía, gracias por enseñarme que la amistad, como la vida, da muchas vueltas. Lucía, gracias por demostrarme que, aunque seamos un desastre y hablemos mucho menos de lo que deberíamos, todo sigue igual. Laura, gracias por ser como eres, por acogerme en Rennes cuando ni siquiera me conocías, por volver a acogerme en Estrasburgo cuatro años después y por demostrarme que la amistad no entiende de distancias.

Pero, sin duda, hay dos personas que merecen una mención especial, y a las que podría agradecer tantas cosas que se me hace difícil encontrar las palabras adecuadas. Laura, haber crecido contigo es uno de los regalos más bonitos que he podido tener. Gracias por estar siempre, siempre ahí, por no perder la sonrisa, por ser incondicional. Mery, gracias por no rendirte conmigo, gracias por apoyarme siempre, incluso cuando no te dejaba hacerlo. Gracias por empujarme a ser mejor, gracias por estar en las buenas y en las malas, y sobre todo en las peores. Y gracias por estos últimos años, porque sé que la tesis la has sufrido casi tanto como yo. Chicas, gracias, gracias y gracias, porque sin vosotras no habría llegado hasta aquí. Gracias por ser mis compañeras de viaje todos estos años.

Por supuesto, mi agradecimiento más especial va para mi familia. Abuela, abuelo, tío, tía, Patri, David, gracias por acompañarme siempre, por compartir mis alegrías como si fueran las vuestras. Gracias a todos por vuestra fe inquebrantable, por confiar en mí más de lo que yo he confiado nunca.

Cristi, gracias por ser mi ejemplo a seguir, por apoyarme y aconsejarme siempre, pero sobre todo gracias por brillar con luz propia y por ser la alegría de la casa, por sacarme siempre una sonrisa. Y Jose, gracias por hacerla brillar aún más, por enseñarnos siempre que juntos sois más.

Mamá, papá, me cuesta mucho resumir lo que os quiero decir, porque no hay palabras suficientes para expresar todo lo que os tengo que agradecer. Gracias por ser mi pilar fundamental, por vuestro apoyo incondicional, por vuestra infinita paciencia y por aguantarme hasta cuando ni yo misma lo hago. Gracias por enseñarme a esforzarme siempre, a seguir, a no rendirme, a luchar por lo que quiero. Soy lo que soy gracias a vosotros, y os debo todo. Sin vosotros, no estaría aquí. Estoy orgullosa de ser vuestra hija. Os quiero muchísimo.

Y, desde hace mucho tiempo, sabía que esto no podía terminar de otra forma más que dándote las gracias a ti. A ti, que has sido mi compañero, que has sido mi amigo, y que ahora lo eres todo. A ti, que me has enseñado lo que es el apoyo incondicional, que me has animado a seguir cuando quería tirar la toalla, que has hecho hasta lo imposible por sacarme todos los días una sonrisa. A ti, que has aguantado mis días malos y mis días peores, cuando todo era negro. A ti, que nunca, ni por un momento, has dudado de mí, y te has enfadado cuando yo dudaba de mí misma. A ti, que me has visto reír, llorar, querer dejarlo todo y volver a empezar, todo en el mismo día. Estoy completamente segura de que, sin ti, no estaría escribiendo esto. Por todo ello, y por mucho más, sólo puedo decirte gracias. Te quiero, Dani.

Resumen

Hoy en día, el aumento de la complejidad de los algoritmos, junto con la ingente cantidad de datos que tienen que procesar y los requisitos de tiempo de ejecución, que cada vez son más estrictos, hace necesaria la utilización de plataformas multinúcleo y, por tanto, el uso de técnicas de paralelización. Todo esto ha contribuido en gran medida a que cada vez sea más difícil desarrollar aplicaciones que cumplan estos requisitos y que, además, usen de forma eficiente los recursos disponibles en la plataforma destino, lo que implica, a su vez, un incremento en el tiempo de desarrollo.

En los últimos años se han propuesto varias metodologías de trabajo como solución a esta problemática, siendo algunas de las más utilizadas el diseño conocido como *Y-chart*, el modelado basado en flujo de datos –del inglés, *dataflow*– y las técnicas de paralelización automática. El diseño *Y-chart* tiene como objetivo aislar la aplicación de la arquitectura, modelando cada una de ellas por separado y combinándolas después, usando para ello un conjunto de directrices establecidas por los desarrolladores. Por otro lado, el modelado basado en flujo de datos consiste en especificar aplicaciones a través de Modelos de Computación –del inglés, *Models of Computation* (MoCs)–, cuyas características son especialmente útiles para exponer el paralelismo estructural de la aplicación. Finalmente, las técnicas de paralelización automáticas se basan en simplificar el proceso de paralelización de las aplicaciones, de tal forma que la dedicación de los desarrolladores a esta tarea pueda minimizarse.

La combinación de *Y-chart* y modelado basado en flujo de datos dio como resultado una metodología donde, utilizando los mencionados MoCs, cada aplicación se modela como un conjunto de tareas –o actores– que intercambian paquetes de información –o, del inglés, *tokens*–, de tal forma que el mapeado y la planificación de la ejecución de dichas tareas aproveche todos los recursos disponibles en la plataforma destino. Aún así, este proceso conlleva todavía un alto grado de implicación por parte de los desarrolladores, ya que dicha planificación necesita ser refinada iterativamente hasta que se alcanza una situación en la que todos los recursos se utilizan de forma eficiente. Esto se debe, entre otros motivos, a que estas tareas son consideradas las unidades funcionales de esta metodología, y por tanto se comportan como cajas negras; esto significa que su contenido depende exclusivamente de los desarrolladores y que, por tanto, su carga computacional puede no estar correctamente balanceada.

Para abordar dicha situación, esta tesis propone diseñar una metodología mejorada que, aunando las tres metodologías descritas anteriormente, optimice aún más el rendimiento de las aplicaciones. Esta metodología tiene como objetivo, por tanto, simplificar el proceso de paralelización y distribución de las aplicaciones sobre arquitecturas multinúcleo, pudiendo utilizarse con un amplio abanico de

aplicaciones. Para ello, se propone combinar una metodología de diseño *Y-chart* basada en flujo de datos con las posibilidades de optimización que ofrece el modelo poliédrico, que se basa en la aplicación de transformaciones agresivas para maximizar el rendimiento de las aplicaciones de forma automática.

Además, otro requisito que ha ganado importancia en los últimos años es que las aplicaciones puedan reaccionar ante cambios de contorno que se produzcan en tiempo de ejecución. Por tanto, sería recomendable que la nueva metodología pudiera garantizar el cumplimiento de este requisito.

Considerando todo lo anterior, se han seleccionado dos herramientas del estado del arte para que actúen de pilares de la metodología propuesta en esta tesis: por un lado, *Parallel Real-time Embedded Executives Scheduling Method* (PREESM) –y su planificador dinámico, *Synchronous Parameterized and Interfaced Dataflow Embedded Runtime* (SPiDER)–, que es un entorno de desarrollo basado en *Y-chart* y flujo de datos; por el otro, *Automatic Speculative Polyhedral Loop Optimizer* (Apollo), que es una herramienta de paralelización automática basada en el modelo poliédrico. Estas herramientas se han elegido porque cumplen el requisito de detectar cambios en tiempo de ejecución y actuar en consecuencia: PREESM a través de SPiDER; y Apollo porque es una de las pocas herramientas existentes en la actualidad que extiende la aplicación del modelo poliédrico al tiempo de ejecución.

La combinación de estas dos herramientas abre la puerta, a su vez, a un amplio rango de oportunidades, puesto que las características intrínsecas a la naturaleza de los MoCs basados en flujo de datos pueden incluirse dentro de Apollo de tal forma que: (i) el comportamiento de esta herramienta se adapte al flujo de datos, y (ii) se exploren las nuevas oportunidades de optimización que surgen como consecuencia de considerar estas características, y que por tanto estaban fuera del alcance de Apollo hasta ahora. Por tanto, Apollo se ha extendido para tener en cuenta dichas características, y gracias a ellas su gestor dinámico ahora incluye un sistema de versionado –en inglés, *multiversioning*– capaz de evaluar varias transformaciones para optimizar el código y elegir la que mejores resultados arroje en función del contexto de la aplicación, que viene definido por características como la cantidad de datos a procesar o la plataforma destino.

Una vez extendida la herramienta poliédrica para contemplar las características de flujo de datos y aprovecharse de ellas, el siguiente paso es embeber esta herramienta dentro del entorno de desarrollo *dataflow* con el objetivo de combinar las múltiples posibilidades de optimización de aplicaciones que estas herramientas ofrecen. Por consiguiente, Apollo se ha embebido dentro tanto de PREESM como de SPiDER, estudiando las diferentes opciones existentes para combinar sus ventajas de tal forma que se obtenga el máximo beneficio de ellas. De esta forma, aparecen nuevas optimizaciones aplicables dentro de cada actor, haciendo posible la explotación simultánea del paralelismo inter-actor –o paralelismo de datos– y de las optimizaciones intra-actor.

Como se ha mencionado antes, una de las principales ventajas de una metodología como la descrita en este documento es que pueda usarse en un amplio espectro de aplicaciones. En este sentido, se ha construido una batería heterogénea de aplicaciones de referencia –del inglés, *benchmark suite*– para validar dicha metodología. Estas aplicaciones han sido extraídas de algunas de las colecciones de aplicaciones más usadas en ambos mundos –poliédrico y *dataflow*–. Por un lado, del dominio poliédrico se han elegido *PolyBench* y *ApolloBench*: la primera colección se ha elegido porque es la más usada por todos los desarrolladores del dominio poliédrico, mientras que la segunda se ha escogido porque es la usada por los desarrolladores de Apollo para validar la versión original de la herramienta, y contiene algunas características específicas que no se contemplan en *PolyBench*. Por otro lado, en el dominio *dataflow* no existe una colección uniforme de aplicaciones de referencia, sino que las aplicaciones varían en función del MoC utilizado; por tanto, para validar esta metodología se han elegido tres de las aplicaciones más usadas por los desarrolladores de PREESM y SPiDER para validar nuevas funcionalidades en la herramienta.

Con respecto a la evaluación de la metodología propuesta con las aplicaciones de referencia extraídas del dominio poliédrico, los resultados obtenidos muestran que, tal y como se esperaba, la versión extendida de Apollo mejora considerablemente los resultados proporcionados por la versión original para aplicaciones que simulan un comportamiento *dataflow*; en concreto, cabe mencionar que estos resultados destacan especialmente cuando la cantidad de datos a procesar es pequeña. Además, comparando los resultados obtenidos con los proporcionados por Clang, se observa que el rendimiento también se incrementa, en este caso en mayor proporción cuando la cantidad de datos a procesar es alta, llegando a obtener aceleraciones de $59\times$ y $161\times$ para *PolyBench* y *ApolloBench*, respectivamente. De la evaluación de estas aplicaciones también se han extraído conclusiones muy interesantes con respecto a la transformación elegida por el sistema *multiversioning* en cada caso: se ha podido comprobar que, como se esperaba, la transformación evaluada tiene un impacto directo en el tiempo de ejecución, ya que los resultados obtenidos con esta transformación pueden ser beneficiosos o no dependiendo de la aplicación, la cantidad de datos a procesar o la plataforma destino, mostrando así los beneficios que aporta un sistema *multiversioning*. No obstante, se debe resaltar que dichos beneficios llevan asociada una penalización inicial en el tiempo de ejecución, ya que este sistema depende de una fase de entrenamiento; a pesar de ello, la naturaleza iterativa de las aplicaciones basadas en flujo de datos hace posible la neutralización de esta penalización al cabo del tiempo.

Además, la evaluación de la metodología con las aplicaciones extraídas del dominio *dataflow* tiene como objetivo principal estudiar las ventajas e inconvenientes asociados a otorgar la capacidad de paralelización a una herramienta u otra. Los resultados obtenidos muestran que las tres aplicaciones seleccionadas permiten una buena representación del problema: una de ellas presenta un mejor rendimiento cuando es PREESM quien gestiona la paralelización – $9\times$ con PREESM, $4\times$ con Apollo y $6\times$ con la combinación de ambas, comparando siempre con Clang–; otra

presenta aceleración sólo cuando es Apollo quien paraleliza –en torno a 5×–; por último, la tercera presenta tiempos de ejecución mucho más rápidos cuando ambas herramientas se reparten las tareas –14× con PREESM, 75× con Apollo y casi 100× con la combinación–. Esta última aplicación se ha utilizado también para validar las modificaciones añadidas en SPiDER, demostrando que la metodología propuesta es capaz de detectar cambios en tiempo de ejecución y reconfigurar el comportamiento del sistema *multiversioning* de acuerdo con estos cambios, obteniendo aceleraciones similares a las alcanzadas de forma estática –cercanas a 100×.

Por último, se va a utilizar la metodología propuesta en este documento para implementar la aplicación elegida como caso de uso real para esta tesis; el objetivo de este estudio es caracterizar su aplicabilidad a través de una aplicación utilizada en la actualidad. Este caso de uso se ha extraído de un algoritmo complejo usado para detectar tumores cerebrales en tiempo real mientras la intervención quirúrgica se está llevando a cabo. En concreto, la parte del algoritmo seleccionada como caso de uso es la etapa de clasificación supervisada del algoritmo híbrido específicamente diseñado para esta tarea, ya que es la etapa que necesita más recursos de computación y, por tanto, la que más puede beneficiarse de técnicas de optimización automática. Los resultados obtenidos muestran que, efectivamente, las aceleraciones más altas aparecen cuando se combina el potencial de optimización de ambas herramientas, alcanzando aceleraciones de 4× con respecto a Clang. Además, los resultados obtenidos con esta metodología demuestran ser competitivos con estudios del estado del arte, en los que se ha acelerado este caso de uso utilizando para ello plataformas de altas prestaciones –del inglés, *High Performance Computing* (HPC)– y explotando la paralelización de estas plataformas de forma manual, con el consecuente incremento del tiempo de desarrollo que ello conlleva.

En conclusión, se ha probado que la metodología propuesta en esta tesis conjuga las posibilidades de paralelización que ofrecen los procedimientos *dataflow* con el potencial de optimización ofrecido por las técnicas de paralelización automática, ya sea explotando las ventajas de cada mundo por separado cuando uno de ellos no es capaz de incrementar el rendimiento, o aunando el potencial de dichos dominios cuando el problema en cuestión es susceptible de ser optimizado por ambos.

Abstract

Nowadays, the increase in the complexity of algorithms, the growth of the volumes of information to process and the tightening of the performance requirements have contributed to the need for multicore architectures and parallel processing. These challenges have led to widening the gap between application development productivity and platform usage complexity. In the last few years, many methodologies have been proposed to close this productivity gap, like Y-chart strategy, dataflow-based design or automatic parallelization techniques. Y-chart strategy isolates the application from the architecture, modeling each one separately and merging them afterwards under a list of user-defined constraints; dataflow-based design proposes modeling applications using dataflow Models of Computation (MoCs), which are specifically well-suited for exposing their structural parallelism; automatic parallelization techniques parallelize applications with very small effort from developers side. The combination of the first two resulted in a methodology where dataflow-based applications are modeled as a set of tasks –or *actors*– exchanging pieces of information –or *tokens*–; these tasks can be automatically deployed onto multicore platforms, mapping and scheduling them so they can make use of all the available resources of the target architecture. Nonetheless, this process is yet arduous from developers side: since the computational load of actors may considerably vary, conceiving an efficient mapping and scheduling of them results in an iterative process. These actors are the functional units of these models, and thus they are considered as black boxes –primitives of the system–, so up until now there is an open issue to solve if their computational load is not well balanced.

To tackle this situation, this PhD aims at devising a methodology to further optimize applications throughput by combining these three methodologies, using it with a wide range of applications and simplifying the process of parallelizing and deploying them onto multicore architectures. To do so, the methodology proposed in this document aims at combining a Y-chart dataflow-based design with the optimization potential offered by the polyhedral model, which is a mathematical framework applying aggressive transformations to maximize applications performance.

As nowadays there is a trend to consider reactivity to changes at runtime as a requirement for many applications, it is desirable for the enhanced methodology to ensure this requirement is fulfilled. Hence, two frameworks are selected as cornerstones of this methodology: Parallel Real-time Embedded Executives Scheduling Method (PREESM) –and its dynamic counterpart, Synchronous Parameterized and Interfaced Dataflow Embedded Runtime (SPiDER)– from the Y-chart dataflow-based domain, and Automatic Speculative Polyhedral Loop Optimizer (Apollo) from the polyhedral one. These frameworks are chosen because they guarantee this runtime reactivity:

PREESM thanks to SPiDER, and Apollo because it is one of the few tools in the state of the art successfully extending the polyhedral model to runtime.

The combination of dataflow and polyhedral domains creates, in turn, a broad spectrum of new opportunities: the characteristics intrinsic to dataflow MoCs can be embedded within Apollo to (i) make it dataflow-based compliant, and (ii) explore new optimization possibilities these characteristics enable, which were out of reach until now. As a result, Apollo is extended to include these two features, providing its runtime manager with a multiversioning system to evaluate several optimizing transformations and choose the best one depending on the application context, which in turn is defined by features as volume of data to process or target architecture.

Similarly, after modifying and extending the automatic optimization framework to include these new functionalities, it needs to be immersed into the dataflow design environment so as to join their parallelization capabilities. Hence, Apollo has been embedded within both PREESM and SPiDER, assessing how to balance the parallelization potential of both frameworks to make the most of them. By doing so, new optimizations are enabled inside actors to exploit intra-actor optimization opportunities together with inter-actor parallelism –i.e., data parallelism.

Since it is desirable for the methodology to be used with a broad range of applications, a heterogeneous benchmark suite is devised to validate the mentioned methodology with applications coming from different domains. To do so, this suite is built by combining some of the most traditionally used benchmark suites in both the polyhedral domain and in the dataflow one. From the polyhedral domain, PolyBench and ApolloBench are chosen, the former because it is the most widely used benchmark suite in the community, and the latter because it was the one used by Apollo developers to originally validate the tool, and it contains several features not represented within PolyBench. From the dataflow domain, due to the lack of uniformity in dataflow benchmark suites, the benchmarks chosen are three of the applications most widely used by PREESM and SPiDER developers when testing new functionalities.

The results obtained from the evaluation of the proposed methodology with the polyhedral benchmarks show that, as expected, the extended version of Apollo outperforms by far the original one for applications simulating a dataflow-based behavior, providing particularly good results for small problem sizes. When compared to Clang, the results demonstrate that the throughput is also improved, especially for the largest problem sizes, where maximum speedups of $59\times$ and $161\times$ are achieved for PolyBench and ApolloBench, respectively. This study shows also interesting results regarding the transformation selected for each case, proving that the transformation has a direct impact on the execution time: depending on the benchmark, problem size or target architecture, the same transformation may provide very good or very bad results, hence illustrating the benefits of a multiversioning system. This study proves that using such a system comes at the cost of adding a non-negligible initial overhead, but that it can be canceled out thanks to the iterative nature of dataflow applications.

Additionally, the main objective of the evaluation of the methodology with the dataflow benchmarks is to assess the advantages and disadvantages of assigning the parallelization capabilities to one framework or the other. The three applications selected prove to be a good representation of this problematic: one of them obtains better results when PREESM parallelizes –9× with PREESM, 4× with Apollo and 6× with the combination, always compared to Clang–; another only obtains speedups when Apollo is in charge –around 5×–; finally, the last one is much faster when the optimizations come from both frameworks –14× with PREESM, 75× with Apollo and almost 100× with the combination–. The latter is also used to functionally validate the changes added within SPiDER, proving the proposed methodology to successfully detect runtime changes and reconfigure the behavior of the multiversioning system accordingly, with similar speedups than those obtained statically –close to 100×.

These two studies ensure the correctness of the proposed methodology, using to that end benchmark suites widely applied in the related domains; nonetheless, so as to guarantee that the proposed methodology fulfills the objectives for which it is envisioned and characterize its applicability, a real-life application is implemented using this methodology, considering it as a real use-case. This application has been extracted from a complex algorithm used to locate human brain tumor boundaries in intraoperative real-time during surgical procedures: specifically, the part of the algorithm selected to conform this use-case is the supervised classification stage of the hybrid classification algorithm specifically designed for this task, as it is the most time-consuming part of the algorithm. The results obtained show that, as expected, the largest speedups are reached when combining the optimization potential of both tools, with speedups close to 4× when compared to the sequential version. These results prove also to be competitive with those extracted from the state of the art, showing similar execution times using fewer Processing Elements (PEs) and applying automatic parallelization techniques instead of manually exploiting the parallelization, which in turn involves a non-negligible increase in the production times.

In conclusion, the methodology proposed in this PhD has been proven to be successful in merging the parallelization potential of dataflow-based specifications with the optimization capabilities of automatic parallelization approaches, which is achieved by either exploiting separately the advantages of each domain when one fails to increase the throughput, or profiting from both potentials in those situations compliant with both worlds.

Contents

Agradecimientos	vii
Resumen	xi
Abstract	xv
Contents	xix
List of Figures	xxiii
List of Tables	xxvii
Acronyms	xxix
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	6
1.3 Thesis layout	7
2 State of the Art	9
2.1 Application modeling	10
2.1.1 Properties	10
2.1.2 MoCs	11
2.2 Application acceleration	15
2.2.1 Levels of parallelism	15
2.2.2 Parallel programming APIs	16
2.2.3 Dataflow-based design	18
2.2.4 Automatic parallelization techniques	20
2.2.5 Overview	26
2.3 Evaluation: benchmarks	27
2.3.1 Dataflow benchmarks	28
2.3.2 Automatic parallelization benchmarks	28
2.4 Evaluation: real use-case	29
2.4.1 Hyperspectral Imaging (HSI)	30
2.4.2 Hyperspectral Imaging (HSI) for cancer analysis	32
2.4.3 Hyperspectral Imaging (HSI) processing methods	36
3 Methodology specification and validation	39

CONTENTS

3.1	Methodology specification	39
3.2	Validation methodology	44
4	Technology framework	47
4.1	Polyhedral domain: Apollo	47
4.1.1	Previous concepts	47
4.1.2	Apollo	52
4.2	Dataflow domain: PREESM and SPiDER	56
4.2.1	PREESM	57
4.2.2	SPiDER	62
5	Dataflow-based runtime polyhedral optimizations	65
5.1	Multiversioning	67
5.2	Evaluation of the best versions	73
5.3	Memoization	78
5.4	Rollback mechanism	80
5.5	Code specialization	83
5.6	Summary	87
6	PiSDF MoC Extension through polyhedral transformations	89
6.1	Polyhedral-dataflow interface	90
6.2	Dataflow preparation	95
6.2.1	Combining parallelization capabilities	95
6.2.2	Feeding dataflow information to the polyhedral domain	98
6.3	Polyhedral preparation	102
6.4	Discussion	107
7	Validation	109
7.1	Functional validation: benchmarks	109
7.1.1	Dataflow-based benchmarks	110
7.1.2	Automatic parallelization benchmarks	111
7.2	Real use-case validation: application	114
7.2.1	HELICoiD project	114
7.2.2	HELICoiD database	115
7.2.3	HELICoiD processing algorithm	116
8	Results	123
8.1	Setup	123
8.2	Functional validation results	127
8.2.1	Polyhedral benchmarks	127
8.2.1.1	Specific setup	128

8.2.1.2	Transformation selection	131
8.2.1.3	Performance evaluation	137
8.2.2	Dataflow benchmarks	157
8.2.2.1	Specific setup	159
8.2.2.2	Static performance evaluation	164
8.2.2.3	Dynamic performance evaluation	171
8.3	Use-case validation results	175
8.3.1	Use-case dataflow implementation	175
8.3.1.1	Implementation details	177
8.3.1.2	Input images	190
8.3.2	Specific setup	191
8.3.3	Performance results	191
8.4	Discussion	200
9	Conclusions	203
9.1	Conclusions	203
9.2	Contributions	206
9.2.1	Dataflow-based runtime polyhedral optimizations	206
9.2.2	PiSDF MoC extension through polyhedral transformations	208
9.2.3	Heterogeneous benchmarking	210
9.2.4	HSI cancer detection	211
9.3	Impact of the thesis	212
9.3.1	Publications and dissemination	212
9.3.1.1	Journal publications	212
9.3.1.2	Conference publications	215
9.3.1.3	Other dissemination channels	217
9.3.2	Research projects	218
9.3.3	Collaborations	219
9.3.4	Open-access products	220
9.3.5	Grants obtained	220
9.3.6	Awards received	221
9.4	Future research lines	221
9.4.1	Runtime extensions of the polyhedral model	222
9.4.2	Extensions to the proposed methodology	223
9.4.3	HSI cancer detection application	224
	Bibliography	225

List of Figures

1-1	Example of a dataflow graph and its associated workload distribution . . .	3
1-2	Overview of the improved design methodology proposed	4
2-1	Example of POSIX threads basic usage	17
2-2	<i>Fork/join</i> execution diagram in OpenMP	17
2-3	Diagram showing the hierarchical distribution of OpenCL	18
2-4	Example of a loop nest and its polyhedral representation	21
2-5	Simplified structure of a hyperspectral cube	31
2-6	Taxonomy of the main research fields of HSI for cancer detection	32
4-1	Example of a loop nest represented as a polyhedron	48
4-2	Loop transformed and polyhedral representation of its iteration domain .	49
4-3	Code and representation of the associated iteration domain: original loop kernel (left) and transformed loop kernel (right)	51
4-4	Example of a TLS system: a) code and b) sequential execution (top) and speculative execution (bottom)	52
4-5	Overview of Apollo	53
4-6	Transitions between behaviors of Apollo	54
4-7	Overview of the code generation flow of Apollo	55
4-8	Succession of chunks, alternating among the different phases of Apollo .	56
4-9	Overview of PREESM framework	57
4-10	Example of a PREESM application graph representing a <i>Sobel</i> filter	57
4-11	Example of a PREESM S-LAM architecture graph denoting a platform with 4 cores	58
4-12	PREESM scenario. <i>Constraints</i> tab example	59
4-13	PREESM workflow	59
4-14	Example of a PREESM application graph representing a <i>Sobel</i> filter with <i>Split</i> and <i>Merge</i> actors to split the processing of the application	60
4-15	Example of a PREESM network graphically exposing the parallelization of <i>Sobel</i> actor	61
4-16	Example of a SPiDER application graph representing a <i>Sobel</i> filter	62
4-17	SPiDER workflow	63
4-18	SPiDER runtime structure	63
5-1	Overview of the improved design methodology proposed in this document to combine the Y-chart design methodology with polyhedral optimizations	66

LIST OF FIGURES

5-2	Detail of the extension of the Y-chart design methodology through a runtime multiversioning system based on polyhedral transformations . . .	67
5-3	Apollo diagram	69
5-4	Overview of the different phases of the dynamic multiversioning system: training phase (left) and operational phase (right)	70
5-5	Diagram of Apollo with multiversioning	71
5-6	Example of different domains: a) rectangular and b) triangular	75
5-7	Generic form of a loop nest of depth 3	76
5-8	Example of a loop nest and its related iterators	77
5-9	Diagram of Apollo with multiversioning and memoization	81
5-10	Behavior of the multiversioning system when a rollback occurs	82
5-11	Complete overview of the dynamic framework including multiversioning, memoization and code specialization: training phase (left) and operational phase (right)	85
5-12	Diagram of Apollo with multiversioning, memoization and code specialization	86
6-1	Overview of the tasks envisioned to embed the polyhedral domain in the Y-chart, dataflow-based design methodology	90
6-2	Example of two instances of the same actor applying the same processing to different input data simultaneously	99
6-3	Overview of the mechanism linking dataflow and polyhedral information	100
6-4	Adaptation of dataflow applications to be compliant with Apollo	102
6-5	PREESM modified workflow	105
6-6	SPiDER modified workflow	106
7-1	HELICoiD demonstrator: (a) block diagram and (b) example of its usage during a neurosurgical procedure at the University Hospital Doctor Negrin of Las Palmas de Gran Canaria, Spain	115
7-2	Block diagram of HELICoiD processing chain	116
7-3	HELICoiD results: representation of the input, the partial results and the final classification provided to the surgeon	117
7-4	Block diagram of HELICoiD supervised classification stage	118
8-1	Overview of the behavior of pcg-Small kernel during training and operational phases of the multiversioning system	154
8-2	PREESM application graph of Sobel filter	159
8-3	PREESM application graph of Stereo	161
8-4	PREESM application graph of the <i>Cost_Parallel_Work</i> actor	162
8-5	PREESM application graph of Matmul	163
8-6	Speedups achieved with PREESMPar, ApolloPar and CombPar for the actors targeted by Apollo in Namaka	169
8-7	Detail of the performance indicators measured with PAPIFY (L1-ICM, L1-DCM and execution time) for Matmul-Large in Namaka with Clang and Apollo	170

8-8	SPiDER application graph of <i>Matmul</i> , where dynamic parameters and actors are denoted with a white circle	171
8-9	Detail of the evolution of the execution time (in seconds) with respect to the iterations of the application graph	174
8-10	Block diagram of HELICoiD hybrid classification framework, where the part conforming the real use-case is highlighted	176
8-11	Top-level SPiDER graph of HELICoiD supervised classification	177
8-12	SPiDER application graph of the <i>HSI_cancer_detection</i> actor	178
8-13	Detail of the actor to pass parameters to lower levels of the hierarchy	179
8-14	SPiDER application graph of the <i>SupervisedClassification</i> actor	180
8-15	SPiDER application graph of the <i>SVM</i> actor	181
8-16	SPiDER application graph of the <i>DimensionalityReduction</i> actor	183
8-17	SPiDER application graph of the <i>ImageCentering</i> actor	184
8-18	SPiDER application graph of the <i>SpatialSpectralFiltering</i> actor	187
8-19	Example of the search window mechanism	188
8-20	Detail of the execution time measured for each invocation of the training phase for <i>ApolloPar</i> in Namaka and Magerit	198
8-21	Comparison of the execution times of <i>CombPar-Oper</i> in Namaka and the implementations presented in [Lazcano'19b]	199
9-1	Overview of the proposed methodology (based on Figure 1-2)	207
9-2	Figure 6-1 (repeated)	208

List of Tables

2-1	Summary of the different MoCs	14
2-2	Summary of the different parallelization methodologies	26
3-1	Summary of the different parallelization methodologies and their evaluation (repeated)	40
3-2	Summary of the different dataflow-based design frameworks	42
3-3	Summary of the different automatic parallelization techniques	43
5-1	Flags to configure the polyhedral transformations applied by Pluto	72
5-2	Information stored for each optimizing transformation	73
5-3	Information stored for each optimizing transformation, evolving from Table 5-2	79
5-4	Information stored for each optimizing transformation, evolving from Table 5-2 and Table 5-3	82
5-5	Information stored for each optimizing transformation, evolving from Table 5-2, Table 5-3 and Table 5-4	85
6-1	Polyhedral-dataflow API	93
6-2	Summary of the different modes defined in Apollo	98
6-3	Information stored for each optimizing transformation, evolving from Table 5-5	101
7-1	PolyBench kernels	112
7-2	Summary of the different sizes predefined by PolyBench	112
7-3	ApolloBench kernels	113
7-4	Characteristics of ApolloBench kernels	113
8-1	List of Pluto flags (repeated)	124
8-2	Pluto flag combinations used for multiversioning	125
8-3	Summary of the different modes defined in Apollo	127
8-4	Optimizations selected by the multiversioning system with mode 2 . . .	132
8-5	Optimizations selected by the multiversioning system with mode 3 . . .	133
8-6	Summary of the number of times each version is selected	134
8-7	Performance results obtained with Clang (in seconds)	138
8-8	Performance results obtained with standard Apollo (in seconds)	139
8-9	Performance evaluation of ApolloBench with mode 2 (in seconds) . . .	143
8-10	Performance evaluation of ApolloBench with mode 3 (in seconds) . . .	144
8-11	Performance evaluation of PolyBench with with mode 2 (in seconds) . .	145
8-12	Performance evaluation of PolyBench with mode 3 (in seconds)	148

LIST OF TABLES

8-13	Performance results obtained for pcg kernel in Magerit (in seconds) . . .	156
8-14	Summary of the different problem sizes for each dataflow application . .	163
8-15	Performance results obtained for the dataflow applications (in seconds) .	165
8-16	Workload distribution of the dataflow benchmarks in Namaka (in %) . . .	167
8-17	Detail of the performance results obtained for the actors in which Apollo is invoked in Namaka (in milliseconds)	168
8-18	Detail of the performance results obtained for Matmul application with SPiDER for both problem sizes and architectures (in seconds)	173
8-19	HSI brain cancer images used during the validation	190
8-20	PCA, SVM and KNN results obtained for the dataflow implementation . .	192
8-21	Performance results obtained for the sequential execution of the use- case (in seconds) and workload associated to each actor (in %)	193
8-22	Configuration of the parameters related to the parallelization	194
8-23	Actor average workload (in %) and possibility to be targeted by Apollo . .	195
8-24	Execution times measured (in seconds) and speedups achieved (in times) for the dataflow implementation of the real use-case	196
8-25	Characteristics of the HPC platforms used in [Lazcano'19b]	199

Acronyms

3D	three-dimensional
5-ALA	5-Aminolevulinic Acid
ADAS	Advanced Driver Assistance Systems
ANN	Artificial Neural Network
API	Application Programming Interface
Apollo	Automatic Speculative Polyhedral Loop Optimizer
AST	Abstract Syntax Tree
BSP	Bulk Synchronous Parallel
CAL	Cal Actor Language
CAMUS	Compilation pour les Architectures Multi-coeurs
Candl	Chunky Analyzer for Dependences in Loops
CD	Compute Device
CERBERO	Cross-layer model-based framework for multi-objective design of Reconfigurable systems in uncertain hybrid environments
CHILL	Composable High-Level Loop
CLooG	Chunky Loop Generator
CNN	Convolutional Neural Network
CPS	Cyber-Physical System
CPU	Central Processing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DAG	Direct Acyclic Graph
DBN	Deep Belief Network
DBT	Dynamic Binary Translation
DCM	Data Cache Misses
DPN	Dataflow Process Network
DSL	Domain Specific Language

LIST OF TABLES

ExanaDBT	Execution-driven Application Analysis and Dynamic Binary Translation
FET	Future and Emerging Technologies
FIFO	First-In First-Out
FP7	7 th Framework Programme
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GeCoS	Generic Compiler Suite
GPU	Graphical Processing Unit
GRT	Global Run-Time
HDL	Hardware Description Language
HELICoiD	HypErspectraL Imaging Cancer Detection
HKM	Hierarchical K-Means
HLS	High Level Synthesis
HPC	High Performance Computing
HSI	Hyperspectral Imaging
ICM	Instruction Cache Misses
IEEE	Institute of Electrical and Electronics Engineers
IETR	Institut d'Électronique et de Télécommunications de Rennes
iMRI	Intraoperative Magnetic Resonance Imaging (MRI)
INSA	Institut National des Sciences Appliquées
I/O	Input/Output
IR	Intermediate Representation
ISL	Integer Set Library
ISODATA	Iterative Self-Organizing Data Analysis
IUMA	Instituto Universitario de Microelectrónica Aplicada
iUS	Intraoperative Ultrasound
JIT	Just-in-Time
KNN	K-Nearest Neighbors
KPN	Kahn Process Network
LLVM-IR	LLVM Intermediate Representation

LRT	Local Run-Time
MAPS	MPSoC Application Programming Studio
MoC	Model of Computation
MPPA-256-N	Multi-Purpose Processor Array
MPSoC	Multi-Processor System on Chip
MRI	Magnetic Resonance Imaging
MV	Majority Voting
NIPALS	Non-Linear Iterative Partial Least Squares
OpenCL	Open Computing Language
OpenHMPP	Open Hybrid Multicore Parallel Programming
OpenMP	Open Multi-Processing
ORCC	Open RVC-CAL Compiler
PCA	Principal Component Analysis
PE	Processing Element
PFG	Proyecto Fin de Grado
PiMM	Parameterized and Interfaced dataflow Meta-Model
PipLib	Parametric Integer Programming Library
PiSDF	Parameterized and Interfaced Synchronous Dataflow
PLATINO	Plataforma HW/SW distribuida para el procesamiento inteligente de información sensorial heterogénea en aplicaciones de supervisión de grandes espacios naturales
PMC	Performance Monitoring Counter
PN	Petri Network
POSIX	Portable Operating System Interface
PPCG	Polyhedral Parallel Code Generator
PPN	Polyhedral Process Network
PREESM	Parallel Real-time Embedded Executives Scheduling Method
QTH	Quartz Tungsten Halogen
RAR	Read-After-Read
RAW	Read-After-Write
REA	Research Executive Agency

LIST OF TABLES

RF	Random Forest
RGB	Red-Green-Blue
ROI	Region Of Interest
RTL	Register Transfer Level
RVC	Reconfigurable Video Coding
SAM	Spectral Angle Mapper
SCoP	Static Control Part
SDF	Synchronous Dataflow
SIMD	Single Instruction, Multiple Data
S-LAM	System-Level Architecture Model
SNR	Signal-to-Noise Ratio
SoC	System-on-Chip
SPiDER	Synchronous Parameterized and Interfaced Dataflow Embedded Runtime
SVM	Support Vector Machine
SynDEx	Synchronous Distributed Executive
TLS	Thread-Level Speculation
TSE	Transformation Space Exploration
ULPGC	Universidad de Las Palmas de Gran Canaria
UPM	Universidad Politécnica de Madrid
VAADER	Video Analysis and Architecture Design for Embedded Resources
VNIR	Visible-to-Near-Infrared
WAR	Write-After-Read
WHO	World Health Organization

1

INTRODUCTION

1.1 Motivation

Nowadays, applications are growing in complexity: the volume of information to process is becoming larger and larger (big data, 4K video...), the complexity of the algorithms to process these volumes of information is also increasing (machine learning, neural networks...), and the performance requirements are being tightened as well (real-time responses). On top of that, some applications are characterized by a strong interaction with both users and the environment, so they need to be reactive and dynamic so as to adapt themselves to mutable requirements [Palumbo'19a]. All these challenges have contributed to the need of parallel processing and multicore architectures, which in turn implies a growth in their usage complexity. Hence, in the last years the gap between platform usage and development productivity has widened.

Exploiting the parallelism of the algorithms so as to efficiently use the available resources of the target architecture while balancing both functional and non-functional requirements is a challenging task, which in turn requires an ever increasing effort from the developers' side. Y-chart design strategy [Kienhuis'01] presents itself as an approach to bridge this productivity gap, proposing a methodology to reduce this effort by isolating the application from the architecture, merging them afterwards under a set of user-defined constraints. In other words, this methodology proposes to have two separate models, one for the architecture and one for the application, and combine them through some user-defined parameters.

Modeling an application implies using a formalism to describe its behavior, so it can be abstracted from the implementation details. As stated by Edwards *et al.* [Edwards'97], the process of designing an application should be based on the use of one or more formal methods to describe the behavior of the system at a high level of abstraction, before any implementation-related decision is taken. Following this approach, the final implementation, derived from the high-level abstraction, will be 'correct by construction'. This approach implies separating the design of the application from the programming language used for its implementation. This is especially interesting if the objective is parallelizing the application: imperative languages, such as C or C++, do not often exhibit the concurrency available in the algorithm; if the design methodology provides a breakdown of the application into subtasks that can then be scheduled onto parallel processors, a more efficient use of concurrent resources can be expected [Lee'87].

This abstraction can be provided by Models of Computation (MoCs) [Savage'98]. A MoC is a set of operational elements that can be used to mathematically describe the behavior of an application, and which can be associated to some properties, as timing or resource requirements. Additionally, a MoC specifies a set of rules to control how applications are executed. These rules and properties provide the theoretical framework to formally analyze the characteristics of applications described using a given MoC. In other words, MoCs are used to increase applications analyzability in such a way that, in design time (when building the application), both the developers and the tools they are using are able to analyze the structure of the application and extract characteristics such as deadlock-freeness, memory usage or latency.

Since the introduction of modern computing systems, a cornucopia of MoCs have been proposed by the scientific community. To devise the proper MoC, one must identify the required properties, so the developers can select the one whose semantics will ease their expression. These properties are usually related to analyzability, determinism, expressiveness, modularity, reconfigurability or parallelism.

Considering this last property, a specific subset of MoCs, known as dataflow MoCs, can be a solution for the application modeling in the Y-chart design strategy, as they are a promising solution to ease the representation of the application parallelism in imperative languages (such as C or C++). A dataflow MoC consists of modeling an application as a graph of functional entities, known as *actors*, that interchange data packets –*tokens*– through First-In First-Out (FIFO) queues, easing the workload distribution by automatically mapping and scheduling the actors onto the available computational resources of the target platform.

However, although the combination of Y-chart design and dataflow MoCs allows a rapid prototyping of applications on complex architectures, it still requires a considerable effort from the developers' side, since the design of the application and its distribution in actors is still up to them. This task becomes especially relevant when considering the resources available on the target architecture: the specification could be too fine-grain, or too coarse-grain, and hence not efficient for the selected target. For that reason, when designing the graph of an application, developers usually follow an iterative process: first, an initial proposal of the division in actors is done and evaluated; most times, this specification is still too coarse-grain and the performance requirements are still far from being fulfilled, so developers need to iterate over the graph, until the granularity of the actors distribution is such that it efficiently uses the resources available in the target platform. Adjusting the graph to the available resources is not a trivial task and there is no automated, extended solution for this, so it normally requires a great effort from developers. Additionally, dataflow graphs are intended to provide a new abstraction layer in such a way that actors become black boxes. Although this is useful for devising the application structure and, therefore, its structural parallelism, it leaves out any potential optimizations that could exist within actors. As a result, exploiting the structural parallelism may not be enough to fulfill the performance requirements, either because of a poor workload distribution or the presence of compute-intensive actors.

To illustrate this drawback, Figure 1-1a provides an example of a dataflow graph. As can be seen, the graph is composed of six different actors, represented as green boxes, which are interconnected with arrows representing the information flow. Each arrow is accompanied by two different numbers, one at the beginning and one at the end, which represent the token production and consumption rates, respectively. There is another number per actor, which represents the number of instances of each actor. When the consumption and production rates are the same, each actor is only repeated once, as happens with all actors but *C*. In this last case, the actors preceding *C* produce twice the number of tokens *C* consumes, so two instances of *C* are generated.

Following the previous description, seven different actor instances have to be scheduled in such a way that their dependences are met: for instance, *C* cannot start its execution until both *A* and *B* have finished theirs, and the same applies to the rest of the actors. Considering all these dependences, a possible workload distribution of this graph for an architecture with two Processing Elements (PEs) is outlined in Figure 1-1b. As can be observed, this kind of graph allows the exploitation of the structural parallelism, since one can easily detect which actors can be executed in parallel and schedule them accordingly. However, this distribution also highlights the problems arising from a poor workload distribution or, in this case, the presence of compute-intensive actors: as depicted in this figure, actor *D* takes considerably longer than the rest of them to be computed, stalling the rest of the processing due to the dependence between *D* and *F*. As a result, *F* cannot start until *D* finishes, causing *PE0* to be idle for a long time, thus reducing the efficiency of the application.

Figure 1-1: Example of a dataflow graph and its associated workload distribution over two PEs.

Nonetheless, this drawback could be tackled down by exploiting some automatic optimization techniques, such as those offered by the polyhedral model. This model is well-known for performing aggressive loop transformations to enhance both data-locality and parallelism on compute-intensive codes, although its scope is traditionally limited to static applications, which is a handicap when runtime reactivity is a requisite. Likewise, tools integrating these techniques usually provide generic solutions, hence losing the optimization possibilities coming from specific solutions.

Consequently, combining a dataflow MoC with polyhedral transformations could lead to an improved design methodology, in which the developers could benefit from all the advantages of the dataflow paradigm and the Y-chart design while balancing their drawbacks with the optimization possibilities offered by the polyhedral model. In this way, the developers would not need to spend time exploiting the intra-actor optimization, thus providing another layer of abstraction to increase their productivity.

Hence, this improved design methodology would be composed of two different levels:

- **Application-level parallelism:** This is the parallelism already offered by dataflow MoCs, which is based on automating the workload distribution of the actors over the available resources.
- **Intra-actor optimization:** This is the new level, which will be based on exploiting the internal optimization possibilities of each actor to try to maximize the application's throughput.

The improved design methodology proposed in this document is outlined in Figure 1-2. This figure represents the Y-chart design strategy, with the separation of application ① and architecture ② modeling, and its successive integration with some user-defined constraints ③ to generate an implementation of the application at compile-time. Then, at runtime, the execution of the application ④ is presented as an iterative process and immersed into the polyhedral domain, which deals with exploring different optimizing transformations that could fit with each actor ⑤, evaluating them ⑥ and taking decisions based on this exploration ⑦.

Figure 1-2: Overview of the improved design methodology proposed in this document.

To demonstrate the advantages of this enhanced methodology, it is going to be applied to a real scenario, which complies with all the challenges for which this enhanced methodology has been devised: large volumes of information, complex algorithms and tight performance requirements. This use-case is the detection of human brain tumors during surgery through Hyperspectral Imaging (HSI).

Nowadays, cancer is one of the leading causes of death around the world. As reported by World Health Organization (WHO)¹, in 2018 it was estimated that this disease was responsible for 9.6 million deaths globally. Regardless of the cancer type, early diagnosis is a key factor in the patient's survival, so there is a growing research interest in this field. Yet, for certain types of cancer, this early diagnosis becomes even more important. When the disease affects crucial organs, as the brain, early diagnosis is a key factor for illness prognosis –i.e., the likely course of a medical condition–. In those cases, apart from identifying it at an early stage, accurately delimiting the tumor to resect as less healthy tissue as possible will have an important impact not only on the patient's prognosis, but also on their quality of life, since removing a larger area of tissue can cause an important damage in the patient, as a permanent disability.

For that reason, in the past years both new and already existing technologies have started to be applied to this field. This is the case of HSI; although it was initially developed for remote sensing and earth observation, its multiple advantages soon led to its application in other fields, such as cancer detection. HSI captures the reflectance of the objects present in the captured scene for a wide range of wavelengths, going from the infrared to the ultraviolet. Hence, in HSI each pixel is formed by the reflectance values of hundreds of spectral bands. Thanks to this information, it is possible to extract a signature that uniquely identifies each object appearing in the image.

Linking this technology to the medical field, it is possible to devise scenarios in which HSI becomes really useful: for instance, if it is applied during a brain tumor resection surgery, it could help surgeons in accurately identifying the margins of the tumor, hence contributing to the patient's prognosis and quality of life. Although brain tumor is bound to be one of the most relevant applications for the reasons aforementioned, this technology can be also applied to detect other types of cancer: tongue, breast, skin, gastric...

Apart from the possibility to extract a spectral signature of the tumors, HSI presents two important advantages: it is a non-contact and non-ionizing technology, which means that it does not need to either radiate the patients or inject them with a foreign substance, which is the case of the existing techniques for delimiting tumors during surgeries. However, HSI presents some important drawbacks:

- **Extremely large volumes of information:** The information provided by HSI leads to extremely large image sizes, which has a direct and negative impact on the time needed to process an image.
- **Complex algorithms:** To process these images, complex algorithms, such as neural networks or machine learning, are usually applied.
- **Intraoperative real-time:** In this context, surgeons should be provided with the results as soon as possible, which implies that the images must be processed in the so-called *intraoperative real-time*, which can be defined as the time needed by the HSI sensor to capture a new hyperspectral cube.

¹<https://www.who.int/health-topics/cancer>

As can be observed, the challenges posed by this application are completely in line with the ones targeted by the enhanced methodology proposed above, so this application is a perfect fit for evaluating it.

In addition, the extension of this methodology to increase the performance of applications will be validated not only with the aforementioned use-case, but also with a set of benchmarks traditionally used in the related domains, which will help in functionally validating the methodology prior to its application in the use-case.

1.2 Objectives

The main goal of the research work described in this document **[OBJ-P]** is to provide a methodology to automatically optimize applications in terms of performance, focusing the efforts on minimizing the time required by the developer to get an efficient implementation. Although the use-case is HSI for brain cancer detection, the objective is to devise a methodology that could be used for a broad range of applications. To achieve this goal, the following objectives have been established:

- **[OBJ-S1]:** To propose a solution to combine automatic optimization approaches, such as polyhedral transformations, with dataflow-based parallelization techniques, so as to provide a new layer of abstraction to the methodology to implement and deploy applications efficiently. To do so, this objective is in turn divided into two subsequent objectives:
 - **[OBJ-S1.1]:** To extend the automatic optimization approach by taking advantage of the characteristics of dataflow MoCs to explore further optimization capabilities, which have been out of reach until now, but become a possibility when feeding the dataflow nature to an automatic optimization framework.
 - **[OBJ-S1.2]:** To embed these new automatic optimization capabilities within a dataflow-based methodology. By doing so, further optimization possibilities are enabled inside tasks, which are currently the functional units of application dataflow-based design, i.e., they are considered black boxes and are not optimized.
- **[OBJ-S2]:** To build a heterogeneous benchmark suite to validate the proposed methodology, conjugating benchmarks traditionally used in both domains to highlight the advantages and disadvantages this methodology may have. The objective of such a benchmark suite is to validate the usage of this enhanced methodology for a wide range of applications.
- **[OBJ-S3]:** To implement the proposed use-case –human brain tumor detection in intraoperative real-time during surgical procedures– using this methodology, so as to characterize its applicability through a real-life application.

1.3 Thesis layout

This document is organized in nine chapters. First, in Chapter 1, the motivations and objectives of this PhD are outlined.

Chapter 2 provides the state-of-the-art related to the concepts, technologies and tools involved in the work presented in this document, which are required to contextualize the solution proposed in this document.

Chapter 3 points out the niche of opportunity detected in the state of the art, introduces the solution that is going to be detailed during this document and describes the validation methodology that is going to be followed.

Chapter 4 outlines the main tools intrinsically linked to the work presented in this document, since they have been used as its cornerstones.

Chapter 5 describes in detail the extensions proposed to the polyhedral framework to make it compliant with dataflow-oriented applications, in such a way that it can take advantage of the optimization possibilities appearing when restricting the applications to those dataflow-oriented.

Chapter 6 details the modifications needed in a dataflow context to combine it with a polyhedral tool to be able to make use of the polyhedral techniques in an automated, transparent fashion.

Chapter 7 presents the strategy followed to validate the methodology proposed, outlining both the benchmarks and the use-case chosen to evaluate the work presented in this document.

Chapter 8 gathers the experimental results: first, the proposed methodology is assessed with some benchmarks related to the polyhedral domain, thus providing the functional validation, and afterwards it is evaluated with the real use-case, which is the HSI human brain cancer detection application.

Finally, in Chapter 9, the main conclusions extracted from this work are analyzed, together with the main future research lines that can be followed to continue with this research line. Additionally, this section also includes: (i) the list of publications derived from this PhD; (ii) the participation on the national and international projects to which the candidate has contributed; (iii) the collaborations carried out during the development of this PhD with other institutions; (iv) the repositories where the open-source contributions of this PhD can be found; and (v) the grants and awards the candidate has received during this PhD.

2

STATE OF THE ART

Chapter 1 introduced the basic ideas upon which this document is going to be built, motivating the research and presenting the main objectives to achieve. Now, the objective of this chapter is to present all the ideas, concepts and technologies related to this research, together with an analysis of what has been done up until now in the topics covered by it.

As exposed in Chapter 1, Y-chart design methodology is a promising approach to bridge the gap between platform usage complexity and application development productivity. It is based on isolating the application from the target architecture, modeling them separately and merging them subsequently through a set of user-defined constraints; in other words, it is based on building two different models, one for the application and one for the architecture, and merging them afterwards to obtain the final implementation.

From the point of view of applications, then, the first step would be to model them, and, more specifically, to be able to choose an appropriate model with which the application is going to be modeled. To that end, Section 2.1 will present the different MoCs used in the literature to model applications, together with some of the most desirable properties a MoC could present.

Once the model has been chosen, the next step would be analyzing how to parallelize the application, since, as aforementioned, the requirements and constraints of applications make compulsory the usage of parallel architectures. In that line, Section 2.2 will first present the different types of parallelism an application may present; afterwards, it will review the different methodologies available to exploit these types of parallelism, exposing their advantages and disadvantages.

Since, as presented in Section 1.1, the main idea of this work is to build an enhanced framework to efficiently exploit the parallelization of applications modeled following the Y-chart design methodology, next step would be devising how to properly evaluate this framework. To do so, Section 2.3 will analyze the different benchmark suites that can be used to functionally validate this kind of framework.

Finally, the last idea is to prove the benefits of the proposed framework with both benchmarks and a real use-case, so Section 2.4 will provide an overview of those selected in this work, studying their scope and the recent literature about the topic.

2.1 Application modeling

As aforementioned, to make an efficient usage of all the resources available in multicore architectures, the target application needs to be parallelized, so it can be split into different regions to be distributed among the different computing processors. However, this process is not an easy task, since one must detect which regions can be executed in parallel and guarantee that those regions are independent from each other, i.e., that they do not share data with other regions, which may imply adapting the code to remove those dependences.

Additionally, there are different levels in which the parallelism of an application can be exploited. This choice is directly linked both to the structure of the architecture, on the one hand, and to the structure of the application, on the other hand. Consequently, devising an efficient parallelization for an application is a task with an ever-growing complexity, since the complexity of architectures and applications is continuously increasing. To ease this procedure, applications can be represented through a MoC, as they increase the analyzability of the application from the design point of view.

As stated in Section 1.1, when using MoCs to design an application, it is crucial to identify its desired properties to choose coherently the most suitable MoC. This section surveys the main characteristics of the most used MoCs: first, the properties used to define and compare MoCs are outlined in Subsection 2.1.1; afterwards, the main MoCs are presented, analyzing them in terms of their properties, in Subsection 2.1.2. The information presented in this section has been extracted from [moc'18].

2.1.1 Properties

This section lists some of the most common properties utilized to compare MoCs, so they can be used afterwards to define the different MoCs that are going to be evaluated.

- **Analyzability:** This property depends on the existence of analysis algorithms to characterize applications, such as those computing worst-case latency or the maximum memory requirements for a given design.
- **Expressiveness:** This feature assesses the level of complexity an application represented with a given MoC can present.
- **Modularity:** A MoC is said to be modular, or hierarchical, if the description of a system using this MoC can be split into multiple, independent modules.
- **Determinism:** If a MoC is deterministic, it means that the output of the system only depends on its related inputs, regardless of external factors.
- **Decidability:** If the schedulability of the applications designed following a given MoC can be demonstrated at compile-time –statically–, then that MoC is said to be decidable [Bhattacharyya'06].

- **Conciseness:** This characteristic represents the ability to express complex system behaviors with a limited description size. This property is very useful when comparing MoCs with similar expressiveness.
- **Compositionality:** It is said that a modular MoC is compositional if the properties of a module described following this MoC can be deduced from those of the submodules composing it [Ostroff'95]. As a result, if all submodules of a given design fulfill a given property, then the whole design –composed by a combination of these submodules– will also fulfill that property by construction.
- **Parallelism:** A MoC is said to be parallel if several modules of the system can be executed independently from each other in a concurrent fashion. On the contrary, a MoC is said to be sequential –non-parallel– if all modules of a system are necessarily executed one after another.
- **Reconfigurability:** A MoC is said to be reconfigurable if the behavior of entire blocks of a system description can be dynamically modified to accomplish future objectives. If a MoC is reconfigurable, then applications using it can dynamically adapt their behavior to external changes.
- **Predictability:** This property is directly linked to the previous one, as it assesses the time elapsed between the reconfiguration of a portion of the system, and the beginning of activity in the already reconfigured part.

2.1.2 MoCs

Considering the characteristics presented above, this section briefly presents the state-of-the-art MoCs used to model applications. They are presented following a bottom-up approach with respect to the level of abstraction, that is, starting with the ones with lower level and finishing up with the ones providing higher levels of abstraction.

Register Transfer Level (RTL)

This well-known model is intended for specifying the behavior and structure of hardware. It is supported by standardized Hardware Description Languages (HDLs), such as VHDL and Verilog, which are parallel languages with explicit parallelism and modular representations at behavioral level. However, although very precise, these languages are too low-level for simulating large systems. They have been traditionally used as an entry point for small to medium size hardware modules, but with the arrival of High Level Synthesis (HLS) tools, there is a tendency to relegate them from being the conventional entry point to being just a requirement for hardware fabrication.

Regarding the properties of this MoC, it is *parallel*, *modular*, *composable* –as it allows a hierarchical description of its components– and *analyzable*. As it provides an event-driven specification at clock-cycle level, it is *deterministic* and *predictable*.

Synchronous Dataflow (SDF)

SDF [Lee'87] is a MoC in which applications are modeled as a directed graph of computational units, hereafter *actors*, which exchange a fixed amount of data, called *data token*, through a set of FIFO queues. It is mainly used to model stream-based and data-driven applications, mostly at component and function levels.

This model, which belongs to the family of dataflow MoCs, is *decidable*, *deterministic*, *parallel* and displays one of the largest levels of *analyzability* among this family. Thanks to these properties, it is possible to prove at compile-time that algorithms modeled with this MoC are, for instance, deadlock-free. For that reason, it is often used to ensure real-time properties, as throughput or latency; nonetheless, this large degree of analyzability comes at the expense of a limited expressiveness. Additionally, there are neither reconfiguration semantics nor modularity in this MoC.

Parameterized and Interfaced Synchronous Dataflow (PiSDF)

PiSDF was born as the result of extending SDF MoC through the Parameterized and Interfaced dataflow Meta-Model (PiMM) methodology [Desnos'13] to add parameterization and interfaced hierarchy to SDF. As a result, applications modeled with PiSDF are still represented as a directed graph, but apart from actors and FIFOs, hierarchical interfaces, parameters and parameter dependences can also be elements of the graph. Additionally, actors can present two types of ports, which can in turn be defined as input or output ports: data ports, for exchanging information, and configuration ports, for exchanging parameters.

Parameters are generically used to either configure or modify the dataflow specification of an application. In this regard, parameters can change or influence: (i) actor functionality; (ii) consumption/production rates of actor ports; (iii) FIFO delays; or (iv) subsequent parameter values, since they may depend on other parameters. PiSDF introduces two types of parameters: locally static parameters and configurable parameters. The former can only be modified at compile-time, while the latter can be modified at runtime, in each iteration of the graph.

As it is an extension of SDF, PiSDF inherits its properties. Additionally, it incorporates *modularity* and *reconfigurability*, while maintaining the *analyzability* inherited from SDF and increasing the *expressiveness*. For these reasons, PiSDF is mainly used to model stream-based, data-driven and control-driven applications, as long as the number of configurable parameters is maintained within reasonable limits.

Bulk Synchronous Parallel (BSP)

BSP [Valiant'90] is specifically well suited for highly parallel architectures, such as Graphical Processing Units (GPUs). When applying this model, applications are divided into multiple phases, called *supersteps*, and several components, hereafter *agents*. An agent can communicate with another agent via a so-called router, which allows the former to access the memory of the latter. As a result, the execution of the application when modeled as a BSP is composed of a set of supersteps composed of processing efforts, global synchronization and remote accesses.

This MoC is *decidable* and promotes *parallelism*. However, as the division of the application into supersteps is performed statically, it presents limited *conciseness*, *reconfigurability* and *expressiveness*.

Kahn Process Networks (KPNs)

KPN [Gilles'74] is mainly used to describe signal processing systems in which processes executed sequentially or in parallel incrementally transform infinite streams of data. Belonging to the family of dataflow MoCs, KPNs are directed graphs where nodes represent computations and arcs depict message queues connecting the nodes. This MoC was originally designed for modeling distributed systems, but it has been proven to be also suitable for modeling signal processing systems.

Systems modeled with KPNs are *deterministic*, since the tokens produced and consumed do not depend on the execution order; they are *parallel* and *compositional*, since applications are divided into processes that can be executed concurrently; however, they can not be proven to be decidable, since it is not possible to tell in a finite time period whether the system will halt in its data streaming.

Polyhedral Process Networks (PPNs) [Verdoolaege'13a] are a specific case of KPNs whose main objective is to automatically derive a parallel specification from a sequential code, provided the fragments of the program that are going to be parallelized satisfy some conditions, namely: the control flow of the program must be static –i.e., not depending on the signals being processed–, all loop bounds, conditions and array index expressions can be expressed using affine constraints, and the functions called within that fragment must be pure. These requirements ensure that all the information related to that program can be represented using mathematical relations belonging to the polyhedral model [Bondhugula'08, Feautrier'11], which will be further described in Subsection 2.2.4.

Dataflow Process Networks (DPNs)

DPN [Lee'95] is a dataflow MoC where functional processing units –*actors*– communicate through unidirectional, unbounded FIFOs. To know when an actor has to be executed, each one has a set of firing rules detailing the amount of data –i.e., tokens– required at their input ports to trigger the processing. Each time an actor is fired, tokens are consumed from the input ports, and produced at the output ones.

DPN is the most *expressive* MoC among all those belonging to the family of dataflow MoCs; this high level of *expressiveness* comes at the cost of analyzability: with DPN, any application can be modeled, deterministic or not, so applications could be very hard to analyze; this *non-deterministic* nature comes together with *non-decidability*. For those reasons, DPN MoCs are usually chosen for modeling streaming applications with intensive computation, data locality and task parallelism, as audio and video coding.

Petri Networks (PNs)

PNs [Giua'07] were first introduced by Carl Adam Petri in the early 1960s as a bipartite weighted directed graph with two types of vertices: places –represented with circles– and transitions –represented with rectangles or bars–. How the net evolves and is executed depends on a set of rules regulating the activation of transitions and data transfers. This MoC is well-suited for modeling event-driven systems, as they can capture characteristics such as concurrency, deadlocks or asynchronous operations.

PNs have been proven to be a very useful visual tool in the design and analysis phase. They are *modular* and *parallel*; on the contrary, they are not *deterministic*, because if several transitions are enabled simultaneously, any of them can be fired. Additionally, an enabled transition is not guaranteed to fire: it can fire immediately, after any amount of time, or not fire at all.

Summary

To conclude this section, Table 2-1 provides a summary of the different MoCs described, evaluating them in terms of the characteristics exposed in Subsection 2.1.1. As can be inferred from the table, PiSDF presents itself as the MoC conjugating the most interesting characteristics while maintaining a medium abstraction level: it presents a high degree of *analyzability*, *expressiveness* and *conciseness*, together with interesting features related to *modularity*, *compositionality*, *parallelism*, *decidability*, *determinism* and *reconfigurability*. The rest of the MoCs reviewed in this section may overcome PiSDF in certain characteristics, but fail to do so in many others: for instance, BSP presents a better expression of the *parallelism*, but lacks *expressiveness*, *conciseness* and *reconfigurability*.

Table 2-1: Summary of the different MoCs and their evaluation through the characteristics defined to compare them. A blank space indicates that the related MoC does not present any specific attribute for the property.

MoC	Analyzability	Expressiveness	Conciseness	Modularity	Compositionality	Parallelism	Decidability	Determinism	Reconfigurability	Predictability
RTL	+			++	++	++		+		+
SDF	++	-	+	-		+	+	+	-	
PiSDF	++	++	++	+	+	+	+	+	+	
BSP		-	-	+	+	++	+		-	
KPN					+	+	-	+		
DPN	-	++				-	-	+		
PN	+	++	+	+		+		-	+	

2.2 Application acceleration

As already mentioned in Chapter 1, exposing the inner parallelism existing within applications is crucial for devising an implementation that efficiently takes advantage of parallel architectures. For that reason, this topic has been thoroughly studied in the literature [Duncan'90, Kessler'07, Kasim'08, Mishra'11, Belikov'13, Hussain'13, Navarro'14, Gu'19]. As can be observed, many research works have been devoted to this topic; although the state of the art is very large, this section is going to provide an overview of the topics that are more aligned to those addressed in this document, starting with the different types of parallelism an application may have –Subsection 2.2.1–, and following with the different techniques and tools that have emerged to deal with parallelizing applications. To that end, these techniques have been classified into three main groups, from a lower to a higher level of abstraction from the developer's point of view: parallel programming Application Programming Interfaces (APIs) –Subsection 2.2.2–, dataflow-based techniques –Subsection 2.2.3– and automatic parallelization techniques –Subsection 2.2.4.

2.2.1 Levels of parallelism

As stated in Chapter 1, applications are growing in complexity in such a way that the only option for fulfilling their performance requirements is to efficiently parallelize them so as to fully exploit the resources available in multicore architectures. However, before getting to the parallelization process, first the different types of parallelism applications may present must be clearly defined. To that end, this section briefly summarizes the different levels of parallelism that can appear within an application.

- **Instruction-level parallelism:** This is the finest level of parallelism. Basically, if an application has instruction-level parallelism, it means that there are instructions that are independent and, thus, can be executed in parallel. This level of parallelism is usually exploited at compile time by the compiler.
- **Data-level parallelism:** An application is data-parallel if the same operations can be independently applied to several blocks of data. Consequently, this level of parallelism is established by the amount of data that can be processed in parallel.
- **Loop-level parallelism:** If an application is said to be loop parallel, it means that it contains for-loops whose iterations are independent, so they can be executed in parallel. Even if the loop is not parallel apparently, there are many techniques to transform loops so as to make them independent [Feautrier'11].
- **Task-level parallelism:** As already mentioned, applications can be modeled as a set of tasks, which are fractions of programs that may be atomically executed. Hence, an application with task-level parallelism can be represented

with a graph that shows all the tasks and the relations among them, i.e., the communications. The number of tasks composing an application is usually larger than the available PEs, so tasks have to be mapped and scheduled.

- **Pipeline-level parallelism:** Pipelining is a technique frequently exploited in streaming-oriented applications. It consists of dividing a task into disjunctive stages to be executed in an overlapping fashion on several computation cores.

2.2.2 Parallel programming APIs

As established before, during the last few decades both applications and architectures have experienced a considerable growth in their complexity. As a consequence, many research lines have focused their efforts in devising different approaches to efficiently use all the available resources in multicore architectures to accelerate applications.

The first approach reviewed is the usage of parallel programming APIs, which are libraries developed to exploit the parallelization possibilities offered by multicore architectures. Specifically, four APIs are going to be surveyed: Portable Operating System Interface (POSIX), Open Multi-Processing (OpenMP), Open Hybrid Multicore Parallel Programming (OpenHMPP) and Open Computing Language (OpenCL).

POSIX threads [Butenhof'97], widely known as just *threads* or *pthread*s, is a language-independent parallel execution model standardized by Institute of Electrical and Electronics Engineers (IEEE). It enables developers to handle several working flows overlapping in time. Each one of these working flows is denoted as a *thread* and is mainly used to distribute the work among the available PEs of a multicore architecture. To ease its usage, it is implemented as a C library with more than 100 procedures organized into four groups: thread management, synchronization between threads, condition variables and mutexes. Due to the many possibilities this library offers, it has been traditionally used to introduce beginners to parallelization concepts and types, such as data parallelism or task parallelism. As mentioned before, this API is the one providing the lowest level of abstraction, since all the procedures, such as thread creation or synchronization, must be handled manually. For that reason, this library is currently used as the foundation of more complex parallel programming APIs, whose objective is precisely increasing the level of abstraction and, thus, decreasing the amount of effort from developers' side. To illustrate how POSIX threads work, Figure 2-1 provides a basic example of its usage. As can be seen, the process starts with the main thread creating two additional threads. The new threads perform their associated processing in parallel and, when they finish, their execution also finishes. As for the main thread, once it creates the additional threads, it has to wait for those threads to finish, but, in the meanwhile, it can also do some processing on its own.

As a matter of fact, **OpenMP** [Dagum'98] was conceived as a solution to allow developers to abstract themselves from the most complex and painstaking operations when using threads, as the synchronization and concurrency management in multicore architectures. To that end, this new standard provides a set of directives

Figure 2-1: Example of POSIX threads basic usage.

to parallelize C, C++ and Fortran applications and deploy them in shared-memory architectures, following the programming model known as *fork/join* [Chandra'01]. This programming strategy, depicted in Figure 2-2, is based on identifying and creating sequential and parallel regions of code inside each application; in that way, the application always starts being executed sequentially by the main thread, and, whenever the thread reaches a region identified as parallel, a *fork* operation is launched. This operation consists of creating several threads to execute in parallel the code included in the parallel region; when this section ends, the *join* operation occurs and the forked threads are merged back into the main thread, which will continue with the sequential execution of the program until a new parallel region is reached, restarting the whole process. Applying this programming model, OpenMP automates fork and join operations, together with the concurrency management, to make it transparent to the user, thus providing a new level of abstraction.

Figure 2-2: Fork/join execution diagram in OpenMP.

With the proliferation of parallel computing paradigms, the usage of heterogeneous architectures to exploit this parallelization has also spread widely. To handle this heterogeneity, several frameworks have been born, as **OpenHMPP** [Dolbeau'07] and **OpenCL** [Stone'10].

On the one hand, **OpenHMPP** is a programming standard for heterogeneous computing, specifically designed for easing the usage of hardware accelerators in conventional general purpose applications while keeping the code portable; in other words, it aims at integrating the usage of hardware accelerators, instead of porting the

application to use them. It is similar to OpenMP, but specifically designed to deal with hardware accelerators. Consequently, OpenHMPP is a directive-based programming model that provides a syntax for offloading computations on hardware accelerators, while also optimizing data exchanges to/from the hardware memory.

On the other hand, **OpenCL** is a solution to handle this heterogeneity and devise a methodology to execute an application using all the available resources, regardless of their nature, such as Central Processing Units (CPUs), GPUs or Field Programmable Gate Arrays (FPGAs). To achieve this objective, OpenCL proposes a hierarchical distribution where all the resources depend on the *host* unit –usually a CPU–, responsible for dispatching the different tasks over the available accelerators, which are supposed to be heterogeneous. Each one of these accelerators is known as a *Compute Device (CD)*; in turn, each CD can be subsequently composed of one or several *Compute Units (CUs)*, which are the units in charge of executing the parallel tasks. Additionally, one or more CDs can conform a *context*. This structure, depicted in Figure 2-3, simplifies the usage of the available resources within heterogeneous architectures to parallelize the same application. Hence, with this framework, users can choose the most suitable PE when assessing the workload distribution. To support this heterogeneity, OpenCL is built upon its own programming language, and it is up to each manufacturer to provide support for their architectures.

Figure 2-3: Diagram showing the hierarchical distribution of OpenCL.

2.2.3 Dataflow-based design

Once the main parallel programming APIs have been reviewed, the next methodology that is going to be presented is the set of tools that have been born during the last few years with the objective of increasing the level of abstraction to reduce developers' efforts with respect to parallelism exploitation and multicore deployment.

Within this methodology, a growing line has focused its efforts in using the MoCs depicted in Section 2.1, namely those with more parallelization potential, to automate the process of exploiting the parallelism to the point to automatically generating

parallel code, ready to be deployed upon multicore architectures. This line is known as *dataflow-based design*, since, among all MoCs, those with more parallelization potential belong to a subset of MoCs known as dataflow MoCs [Ackerman'79]. Their objective is to explicitly isolate processing tasks from data communication jobs in the application design step. Although this involves an extra effort from developers when designing applications, it considerably simplifies automatic mapping and scheduling, thus helping them during the whole process. This approach has usually been linked to the Y-chart design methodology described before [Kienhuis'01], which isolates the application from the architecture, thus providing another layer of abstraction by allowing users to model applications without considering the target architecture.

Synchronous Distributed Executive (SynDEx) [Grandpierre'99] is one of the first frameworks combining these methodologies. It implements the Y-chart prototyping methodology as follows: the application is represented through a Direct Acyclic Graph (DAG) where vertices denote processing operations and edges represent data dependences, while the architecture is modeled as a set of operators –or computation resources– that are connected through communicators, which in turn deal with accessing both memory resources and the dedicated buses. Following the Y-chart design methodology, the final implementation is obtained by combining these two models under a set of user-defined constraints, thus ensuring their compliance. In the case of SynDEx, these constraints are mainly related to real-time requirements, so the mapping and scheduling are performed through a latency-based algorithm [Sorel'05].

MPSoC Application Programming Studio (MAPS) [Ceng'08] is a proprietary dataflow framework that allows the automatic deployment of applications modeled with the KPN MoC over a heterogeneous Multi-Processor System on Chip (MPSoC). It takes as inputs the sequential code and a description of the target architecture with the objective of linking the parallelism possibilities of the architecture with the parallelism of the application. To do that, in turn, it extracts data and control information and extrapolates it to different levels of parallelism, such as data-, task- and pipeline-level.

Open RVC-CAL Compiler (ORCC) [Yviquel'13] is another dataflow-based framework developed with the objective of generating parallel codes from dataflow specifications. To generate this dataflow specification, ORCC uses its own specification language, Reconfigurable Video Coding (RVC)-Cal Actor Language (CAL) dataflow language, and follows the DPN dataflow MoC, which was described in Section 2.1. Since one of the characteristics of this MoC is its lack of determinism, this implies that the applications modeled with ORCC are non-deterministic. On the contrary, an important advantage of this framework is that it is open-source.

Parallel Real-time Embedded Executives Scheduling Method (PREESM) is also an open-source, dataflow-based, Y-chart prototyping framework [Pelcat'14], which relies on using models to describe both the application and the target architecture separately. To model the application, PREESM uses the PiSDF MoC, which extends the SDF MoC by adding two important characteristics, namely hierarchy and parameterization. Likewise, a System-Level Architecture Model (S-LAM) is used to model parallel architectures as a set of PEs exchanging data through data links and communication

nodes. Once both the application and the architecture have been described following their respective models, the final implementation combines them through a set of user-defined constraints. Due to the characteristics of PiSDF, any application modeled with PREESM is guaranteed to be deadlock-free by construction.

Synchronous Parameterized and Interfaced Dataflow Embedded Runtime (SPiDER) [Heulot'14] was developed as an extension of PREESM to support runtime reconfiguration of PiSDF applications. While using the same framework, this extension allows the user to define dynamic parameters within the application graph. These dynamic parameters may change their value during the execution of the application and may affect its structure, so the mapping and scheduling must be remade should a change in those parameters is ever detected. To do so, PREESM is provided with a runtime manager, known as SPiDER, which is divided into two main parts: a Global Run-Time (GRT), which is responsible for orchestrating the execution, deploying the different actors conforming the application over the PEs available in the target architecture; and as many Local Run-Times (LRTs) as physical PEs, which deal with executing actors and, if necessary, sending back the updated values of the dynamic parameters to the GRT so it redistributes the application according to these new values.

PN [Verdoolaege'07a] is a compiler included within the Compaan tool [Kienhuis'00] to ease the task of partitioning applications and synchronizing different partitions to be executed in embedded MPSoCs. As said in [Meijer'10], it derives PPNs, which are a specific case of KPNs, from sequential nested loop applications, with the objective of identifying possible reuses of channels between nodes to effectively reduce the communication volume by eliminating those redundancies. Although PN is focused on just deriving PPNs from sequential programs, one of its main advantages is that the resulting PPN is compliant with the polyhedral model, which performs aggressive optimizations in the code and will be fully described in Subsection 2.2.4.

2.2.4 Automatic parallelization techniques

In the field of parallel computing, automatic parallelization of sequential programs has always been the ultimate goal: devising methodologies, techniques and frameworks that can take a sequential application and automatically generate parallel code, efficiently distributed to take full advantage of the resources available in the target architecture and with as less effort from the developer's side as possible. However, despite decades of efforts in this line, automatic parallelization has had limited success, and full automatic parallelization remains a challenge [Shen'13]. The reasons for this technique to still remain a challenge are numerous, namely: the variability of architectures, applications and parallelism types, the need for complex program analysis and the lack of crucial information during compilation [Fox'14].

Although full automatic parallelization remains a challenge, several research lines have contributed to improving its quality. One of the main targets for automatic parallelization and optimization techniques is loop parallelization, as, in general lines, a considerable part of the execution time of applications is spent in executing some

sort of loops. This technique tries to automatically pipeline or vectorize the loop execution to reduce its processing time. To do so, the loop is transformed to: (i) expose a possibly parallel inner loop well suited for pipelining or vectorization; (ii) undo data dependences preventing a loop from being parallelizable, so it can be exposed and parallelized; and (iii) promote data locality to improve memory accesses.

In this line, the polyhedral model [Bondhugula'08, Feautrier'11] is a powerful geometrical and mathematical framework for analyzing and optimizing loop nests. In the context of High Performance Computing (HPC), it has been studied for decades, providing precise and concise representations of loop nests. The polyhedral model was born as a solution to the limitations of the Intermediate Representation (IR) used in compilers. In traditional intermediate program representations such as Abstract Syntax Trees (ASTs), each statement appears only once, even if executed multiple times inside a loop, impeding to reason about individual statement iterations. On the contrary, the polyhedral model represents individual statement iterations as integer points inside a polyhedron –as shown in Figure 2-4, which provides a toy example to illustrate this description–, easing the analysis and transformation of loops.

Figure 2-4: Example of a loop nest and its polyhedral representation. Based on a figure from [Yuki'13].

To be compliant with the polyhedral model, a loop must fulfill the following requirements: (i) it has to iterate according to a unique index variable, whose bounds are affine expressions of the enclosing loop indices –i.e., they can be expressed as a linear function of the loop indices, plus a constant–; and (ii) the memory instructions are limited to access simple scalar variables or multi-dimensional array elements referenced using affine expressions on the enclosing loop indices. Such loop nests are analyzed precisely with respect to data dependences that occur among the statements and across iterations. Hence, advanced optimizing transformations are proven to be semantically correct by preserving the dependences of the original program.

Nonetheless, although very powerful, this model has been traditionally restricted to a class of compute and memory intensive codes that can be analyzed accurately and transformed only at compile-time. Yet, most legacy codes are not amenable to this model due to indirect or pointer-based accesses to static or dynamic data structures, which prevent a precise dependence analysis to be performed statically. For these reasons, even though the model is powerful, its applicability is limited.

Anyhow, codes that do not exhibit characteristics suiting the polyhedral model at compile time may still be compliant with it, although this compliance can only

be detected at runtime. Targeting such codes for automatic optimization imposes to immerse this model in the context of speculative and dynamic parallelization. Speculative parallelization [Steffan'98], or Thread-Level Speculation (TLS), is also a well-known technique in code optimization and automatic parallelization, which consists of optimistically executing regions of code in parallel, assuming no dependences are violated. While these regions are executed in parallel, an underlying mechanism monitors every memory access ensuring no data races occur and, if so, halting all the involved threads and initiating a rollback to get back to a previous state.

As stated before, the polyhedral model has been widely studied and applied to automatically parallelize applications, but its scope has been mainly limited to applications in which all the parameters are known in compile time. Nevertheless, in recent years there has been some efforts to extend this model to codes compliant with it at runtime. Consequently, the remaining of the section presents an overview of the state-of-the-art polyhedral tools, starting with those applicable to compile-time codes, and finishing up with the tools that extend the model to runtime.

Compile time

Applying the polyhedral model to optimize the performance of applications is a topic thoroughly studied in the literature, so there are numerous frameworks that apply it. In this section, the most relevant ones are highlighted.

Pluto [Bondhugula'08] is an automatic source-to-source transformation framework to optimize C/Fortran applications both in terms of parallelism and data locality. It automatically generates OpenMP parallel code from sequential program sections, to which a wide range of polyhedral transformations can be applied. It can be used as a standalone tool, or as a library. Pluto accepts as input an OpenScop representation [Bastoul'11] of the source code, which is a portable format for representing loop nests in a polyhedral model fashion, and whose objective is to simplify exchanges among tools using this format. One of its main advantages is that it extends the scope of tools like Pluto, so they can be applicable not just to C/Fortran code, but to any input language from which an OpenScop representation can be extracted.

Polly [Grosser'12] is a polyhedral compiler built on top of LLVM which automatically detects and extracts the parts of an application that are compliant with the polyhedral model –Static Control Parts (SCoPs)– in a syntactically transparent, language-independent way. These sections, represented using LLVM Intermediate Representation (LLVM-IR), are then transformed, either manually or through Pluto. Polly includes too Single Instruction, Multiple Data (SIMD) and OpenMP code generation to take advantage of both already-existing and newly-exposed parallelism. Polly follows a three-step approach: first, it detects those parts of the application that are susceptible to be optimized and converts them into a polyhedral representation; afterwards, it optimizes this representation; finally, it regenerates the optimized code, detecting parallel loops and implementing them as SIMD instructions or OpenMP parallel loops. These three steps are implemented as a set of LLVM-IR passes that have been grouped into front end, middle end and back end.

GRAPHITE [Trifunovic'10] is one of the first polyhedral tools integrated into a production compiler, as it is implemented as a pass in the GNU Compiler Collection (GCC) compiler. As Polly, GRAPHITE is independent of the source language, as it extracts the polyhedral representation from GCC low-level intermediate representation, GIMPLE. Also as Polly and many other polyhedral source-to-source compilers, it follows a three-stage process: (i) extract the polyhedral representation from GIMPLE, (ii) apply various optimizations, and (iii) regenerate the GIMPLE intermediate representation with the polyhedral transformations already embedded. Although GRAPHITE is already integrated into GCC, it still presents two relevant shortcomings: first, SCoPs are detected in an unstructured way, which effectively limits the size of the code to optimize; secondly, it only accepts memory accesses to be represented as array accesses, and pointer accesses are not supported.

Composable High-Level Loop (CHiLL) [Chen'08] is a high-level, script-based, source-to-source translator that tries to improve the performance of nested loops written in C, C++ or Fortran. CHiLL works following a script, provided by the user, in which the loops to modify and the transformations to apply are detailed. It can be configured to support NVIDIA Compute Unified Device Architecture (CUDA) compiler, so as to generate source code to be compiled and executed on NVIDIA GPUs.

AlphaZ [Yuki'12] is an open-source polyhedral framework for prototyping analyses and transformations. The main difference with other polyhedral tools from the state of the art is that it allows the user to have full control of the transformations to apply, since its main objective is to develop techniques for automatic parallelizers by providing a system that can be utilized as an engine to test new strategies. The input program must be composed of one or more mathematical equations specifying what needs to be computed. With those equations, it analyzes the program –e.g., it can try to deduce a parallel schedule–, transforms it –always preserving the semantics– and generates code that implements the equations provided as input. The key advantages of this tool are: (i) users can systematically apply sequences of transformations without manual rewriting of the program, (ii) compiler writers can prototype new transformations and/or code generators with much less effort from their side.

R-Stream[®] [Meister'09] is a commercial source-to-source compiler developed by Reservoir Labs¹. It uses the polyhedral model to automatically optimize C programs, translating them into efficient codes targeting accelerators and multicores. As a result, it accepts a sequential C program as input, and can output a parallel version in a variety of formats, including OpenMP, OpenCL, and CUDA. The optimizations applied by this compiler include parallelization, locality optimization, vectorization, high-level software pipelining and explicit memory management, among others. R-Stream[®] is a proprietary technology of Reservoir Labs, and includes patented and patent-pending features to allow rapid compilation in very complex target architectures.

IBM XL [Bondhugula'10] is an optimizing compiler developed by IBM and specifically thought for large and complex C/C++ applications. It presents a polyhedral

¹<https://www.reservoir.com/product/r-stream/>

optimization pass in which several polyhedral transformations are applied to the compiler's high-level intermediate representation to provide automatic parallelization and locality optimization. To do so, the pass automatically generates a script to drive the optimizations, easing automatic and/or user-driven tuning of transformations' parameters, as well as the order in which to apply the transformations.

Generic Compiler Suite (GeCoS) [Floc'h'13] is an open-source framework specifically thought for supplying a highly productive infrastructure to ease the prototyping of hardware design flows. This compiler suite contains a set of passes targeting loop transformations and based on the polyhedral model. These transformations are achieved by using third-party libraries, such as Integer Set Library (ISL) [Verdoolaege'10] –a library for manipulating polyhedral domains– and Chunky Loop Generator (CLooG) [Bastoul'04] –a library for polyhedral code generation–. Although GeCoS was originally developed for hardware design, this framework has been also extended to software. For instance, ALMA European project [Stripf'13] integrated GeCoS into its proposed toolchain to enable efficient application mapping on multicores from a high level of abstraction, namely Scilab [Campbell'06].

Polyhedral Parallel Code Generator (PPCG) [Verdoolaege'13b] is a polyhedral source-to-source compiler that generates code for multicore accelerators with tight memory hierarchies, as GPUs. PPCG is divided into three steps: (i) a polyhedral representation is extracted from C code, (ii) a dependence analysis is performed to create a GPU optimized schedule, and (iii) the polyhedral representation is translated to OpenCL, OpenMP or CUDA code. One of its main advantages is that it implements a multi-level tiling strategy that respects GPUs hierarchical memory architecture.

PENCIL [Baghdadi'15] is a subset of GNU C99 that allows compilers to exploit parallelism and generate highly optimized code for accelerators. It aims at being both a target language for Domain Specific Languages (DSLs) and a portable implementation language for libraries. Additionally, a PENCIL-to-OpenCL backend using PPCG is also provided to generate optimized OpenCL code from a PENCIL specification, so as to provide another layer of abstraction and simplify the process of developing software.

TIRAMISU [Baghdadi'19] is a polyhedral framework developed for generating high performance code for multiple platforms, including GPUs, multicores and distributed machines. One of the main novelties of TIRAMISU is that it presents a new scheduling language with commands to explicitly manage the many complexities arisen when dealing with this kind of systems, allowing a fine-grained control of optimizations. This framework uses a four-level intermediate representation to allow a full separation among algorithms, loop transformations, communication and data layouts. With this separation, the process of targeting different hardware architectures with the same algorithm is largely simplified. This compiler framework has been designed to target applications including image processing, linear algebra, deep learning and stencils.

Runtime

As stated before, despite the potential offered by the polyhedral model, its scope is restricted to codes whose dependences can be analyzed statically, in compile-time. This limitation leaves out, namely, indirect or pointer-based memory accesses, dynamic data structures or dynamic parameter dependences, among many others. However, codes containing these characteristics may still comply with the polyhedral model at runtime, but, evidently, this behavior can only be detected during the execution. Hence, targeting such codes for automatic optimization and parallelization means embedding the polyhedral model in the context of dynamic parallelization. The rest of the section introduces the most relevant tools found in the literature that extend the polyhedral model to runtime, to the best of the author's knowledge.

SPolly [Doerfert'13] is an extension of Polly using runtime information and specializing functions to widen the applicability of the polyhedral model. As stated in [Doerfert'13], the main causes for which Polly rejects a region susceptible to be represented as a polyhedron are aliasing –or overlapping– issues and missing parameter values. Driven by this observation, they propose a set of runtime checks to switch dynamically to specialized variants of a code region, where the preventing issue has been removed. This technique is called *specialization*, and it consists of creating variants of the same code where certain variables are instantiated to a constant value. These versions coexist with the original one and are launched whenever the runtime values allow it. During the first execution of the program, a profile is generated, from which values and aliasing properties are inferred, and specialized variants of the code regions can be created. In this first run, SPolly cannot use the profiling data yet, so it can only apply those speculative transformations for which conclusive runtime checks can be set up. In the second run, the specialized versions are created, based on the profiling information. In these specialized versions, loop bounds and parametric values are substituted by constant expressions, which both enables the representation of those sections as polyhedra and the generation of more efficient code.

Execution-driven Application Analysis and Dynamic Binary Translation (Exa-naDBT) [Sato'17] is a dynamic compilation system that transparently optimizes and parallelizes binaries at runtime through the polyhedral model. This system combines a polyhedral compiler, Polly, with Dynamic Binary Translation (DBT), a technique to migrate on-the-fly code from one architecture to another without compatibility losses [Probst'02], by taking advantage of this process to apply polyhedral optimizations, applying a two-phase dynamic instrumentation and optimization.

- *Transparent application analysis*: It is composed of *profile* and *estimate* stages: the former executes a profiling while the original code is executed to detect hot spots of execution; the latter estimates a kernel which dominates the entire execution, assuming a simple model in which performance gain is achieved if the function optimized is the one containing the first hot spot detected.
- *Transparent optimization*: While executing the unoptimized code, the target kernel function is monitored to check whether it is called again. When that

happens, the second phase begins, which is composed of *translate* and *switch* stages: the former deals with generating optimized code by means of Polly, while the latter switches from the original execution to the optimized one.

Automatic Speculative Polyhedral Loop Optimizer (Apollo) [Caamaño'17] is a framework capable of applying polyhedral transformations on-the-fly to any kind of loop nest –for-loops, while-loops, do-while-loops, goto-loops...–, even if it includes unpredictable memory accesses through pointers or indirections, as long as it is compliant with the polyhedral model at runtime. To do so, at runtime, Apollo builds a prediction model for the target loop nest, which is then fed to Pluto so it determines the loop optimizations to apply. Afterwards, the code generation mechanism feeds the optimizing and parallelizing loop transformations and their associated scan of iteration domains to LLVM Just-in-Time (JIT) compiler in order to generate the binary code, which is then launched. As there is a prediction model involved, this system incurs in speculative parallelization. As a result, some mechanisms must be added to control the behavior of the system if the model fails. To do so, Apollo includes a set of verification instructions, which are in charge of checking continuously that the model is still fulfilled. If these checks fail, Apollo goes back to the point in which the model started to fail and executes the original code for the rest of the chunk so as to overcome the faulty execution point. When this happens, in the next chunk the prediction model is built again, and the process is restarted.

2.2.5 Overview

After reviewing the main state-of-the-art methodologies to parallelize applications, this section is going to be closed with a comparison of them. To do so, two features are considered: the types of parallelism that can be exploited with each of the methodologies presented in this section –described in Subsection 2.2.1–, and the level of abstraction these methodologies provide developers with. This comparison, shown in Table 2-2, is carried out using the three groups in which they have been divided: parallel programming APIs, dataflow-based design and automatic parallelization.

Table 2-2: Summary of the different parallelization methodologies and their evaluation through the characteristics defined to compare them.

	Parallelism					Abstraction level
	Instruction level	Data level	Loop level	Task level	Pipeline level	
Programming APIs	✓	✓	✓	✓	✓	↓
Dataflow-based design	✗	✓	✗	✓	✓	
Automatic parallelization	✓	✓	✓	✗	✗	

As can be seen, parallel programming APIs offer a high level of flexibility, granting the necessary resources to exploit the different types of parallelism. Yet, their level of abstraction is very low: it is up to the developer to efficiently exploit the parallelism.

This task is hard, manual, error-prone and usually very time-consuming. As a result, the trend in research has been to look for methodologies to accelerate this process, while maintaining within reasonable limits the performance achieved. This trend has led to the apparition of methodologies such as dataflow-based design and automatic parallelization, each one increasing the abstraction level with respect to the previous one, respectively. In both cases, each methodology increases the level of abstraction at the cost of narrowing the range of types of parallelism that can be exploited.

Table 2-2 highlights also the motivation of this PhD: by combining dataflow-based design with automatic parallelization techniques, the resulting methodology will present multi-level parallelism capabilities while keeping a high level of abstraction.

In order to build this multi-level parallelization methodology, the tools presented in Subsection 2.2.3 and Subsection 2.2.4 need to be revisited, this time evaluating them in terms of the specific requirements this methodology demands. Since this is out of the scope of this chapter, this study will be carried out in Chapter 3.

2.3 Evaluation: benchmarks

As established in Chapter 1, the objective of the work presented in this document is to build a new methodology to parallelize and deploy dataflow-based applications following the Y-chart design methodology and extending it through automatic parallelization techniques. However, when developing a new methodology, one must provide enough proof to demonstrate its validity, as well as its advantages. The former requires a functional validation of the methodology proposed, while the most effective way to demonstrate the latter is through a real use-case, in which all the benefits of this methodology are highlighted. To that end, this section is going to deal with how to address the functional validation, while Section 2.4 will introduce the state of the art of the use-case selected for this work.

The most used and efficient approach to check the functional validation of either hardware or software is to evaluate it through a set of benchmarks or test suites. Benchmarking is a technique that consists of using a set of standard tests to assess the correctness of systems [Fleming'86], and is also especially useful in research, since it allows a fair comparison with existing techniques or frameworks when new ones arise. This technique is so widely used that specific benchmarks have been developed depending on the nature of the system that is going to be validated. Consequently, the first step is to correctly identify which kind of benchmarks are the most suitable ones to assess the functional validation of the proposed methodology. As stated before, the new methodology is going to be built upon the combination of dataflow-based design and automatic parallelization methodologies. Therefore, the remaining of the section is going to deal with reviewing the state of the art related to the benchmarks most widely used to characterize systems belonging to either category.

2.3.1 Dataflow benchmarks

Although dataflow-based design and dataflow MoCs have been widely used in the last few years to model applications, the diversity among MoCs –and, subsequently, among the characteristics each application must present to be modeled with them– makes difficult to design a standard benchmark suite that can be intensively used to assess dataflow-based design systems. Nonetheless, this section is going to present some of the most cited ones among the related literature.

DFbench² was developed as a solution for this lack of uniformity in dataflow benchmarks, aiming at providing a set of dataflow models of streaming applications with their corresponding implementations. It contains KPN and SDF descriptions of 5 different applications, together with their corresponding implementation in C. These applications are related to the signal processing domain: two decoders, one encoder, an edge and corner detection application and a voice processing application.

DaSH [Gajinov’15] presents itself as the first benchmark suite specifically thought for hybrid dataflow and shared memory programming models. It features 11 kernels with different communication and computation patterns, and includes sequential and shared-memory implementations –based on OpenMP– to ease the comparison between hybrid dataflow and traditional shared-memory implementations.

Although these two benchmark suites demonstrate that there has been efforts to devise a standard benchmark suite for dataflow applications, their usage has been limited to the specific models they target. The tremendous diversity of algorithms and applications led to the appearance of different MoCs to promote different characteristics of applications; thus, it is unlikely that an application can be efficiently represented with all the existing MoCs, and even more, that the mentioned application can be used as a unique benchmark to evaluate the different models. As a result, the most common practice observed in the tools and frameworks presented in Subsection 2.2.3 is to use their own set of applications specifically developed to demonstrate the efficiency and correctness of each system.

2.3.2 Automatic parallelization benchmarks

As mentioned before, automatic parallelization has been a research topic widely explored in the literature, so plenty of benchmark suites can be found. However, as the frameworks presented in Subsection 2.2.4 are mostly related to the polyhedral model, the state of the art presented in the rest of the section is focused on those benchmarks suites specifically thought for evaluating systems based on this model.

In this regard, **PolyBench** [Pouchet’12] is undoubtedly the most widely used and extended benchmark suite for systems compliant with the polyhedral model. This benchmark suite includes numerical computations extracted from several application domains, namely linear algebra, image processing, statistics, physics simulation, data

²<http://www.es.ele.tue.nl/dfbench/>

mining or dynamic programming. PolyBench is currently supported for C and Fortran, and contains 30 different benchmarks.

Rodinia [Che'09] is a benchmark suite specifically developed to evaluate heterogeneous computing, including applications and kernels targeting multicore CPU and GPU architectures. Specifically, this benchmark suite is composed of four applications and five kernels exhibiting various types of parallelism, data-access patterns and data-sharing features. These applications cover a wide range of application domains: data mining, bioinformatics, image processing, etc.

Scimark³ is a Java benchmark suite for scientific and numerical computing that contains 5 kernels specifically chosen to evaluate the performance of systems.

SPARK00 benchmark suite [Spek'08] proposes a set of applications specifically designed to target irregular access patterns, which is one of the main causes of performance degradation in HPC platforms. This suite is meant to be used to evaluate the performance of optimizing compilers, and the irregular access patterns are represented through the usage of pointers, indirection arrays, dynamic loop bounds and runtime dependent if-conditions, which have always been a major challenge for these compilers. Specifically, this suite consists of two subsets of benchmarks targeting array-based and pointer-based applications, respectively, and resulting in a total amount of 10 different benchmarks.

Parboil benchmark suite [Stratton'12] is a set of applications aiming at evaluating the performance of architectures and compilers and coming from diverse application domains, such as image processing, fluid dynamics, astronomy or biomolecular simulation. Additionally, each benchmark includes different implementations to help programmers get the most out of their tools and architectures.

2.4 Evaluation: real use-case

As established in Section 2.3, when designing a new methodology one should perform a double validation: first, the methodology should be functionally validated through a set of benchmarks, preferably with benchmarks used by other related methodologies, so the new one can be easily compared with those already existing in the state of the art; secondly, it should be also validated with a real use-case application, which highlights the benefits the new methodology can provide to a real application.

Section 2.3 has dealt with reviewing the state of the art related to the different benchmark suites used both in dataflow and polyhedral domains, so this section is going to do likewise with the state of the art related to the real use-case that is going to be used to validate the work proposed in this document.

³<https://math.nist.gov/scimark2/>

As advanced in Chapter 1, the real use-case application chosen in this work is hyperspectral imaging for brain cancer detection in real-time. The reasons why this use-case has been selected are twofold:

- First, with respect to the technical requirements, this application presents all the characteristics mentioned in Chapter 1: tight real-time requirements, complex algorithms to process and large volumes of information to analyze.
- Secondly, the application domain has gained momentum in the last years, since many research works are being focused on applying new technologies to the medical field, as is the case of HSI.

The rest of the section is organized as follows: Subsection 2.4.1 provides a review of HSI technology, overviewing the fields to which it has been applied, while Subsection 2.4.2 explores its application to the cancer detection field and, more specifically, to brain cancer detection; finally, Subsection 2.4.3 reviews the different techniques that have been applied to process this kind of images. It should be noted that the information presented in this section has been extracted from a very recent article reviewing the advances of this technology in the medical field [Halicek'19].

2.4.1 Hyperspectral Imaging (HSI)

HSI, also known as image spectroscopy, is a technology that was born as a result of the evolution of earth surface observation and that combines conventional imaging and spectroscopy methods to get both spatial and spectral information from the scene under observation [Kamruzzaman'16]. The main advantage of this technology with respect to conventional Red-Green-Blue (RGB) imaging is that it increases the volume of data captured beyond the three spectral bands of the visible spectrum captured by RGB, including information beyond the capabilities of the human eye [Starr'10]. The specific sensors developed to capture these images can cover a wide range of contiguous spectral bands or wavelengths across the electromagnetic spectrum, and they measure the reflected, absorbed and emitted radiance at certain wavelengths of the materials under observation, so a specific vector of radiance values, hereafter *spectral signature*, can be extracted for each pixel of a hyperspectral image. With this signature, image processing algorithms can automatically identify and distinguish the different materials captured by the sensor at each pixel [Manolakis'02]. To do so, these algorithms rely on the proven fact that the different molecular compositions of each material existing in the nature react in different ways to the incident light [Kamruzzaman'16]. This means that the spectral signature can identify each material in an unambiguous way, and hence can be used by the aforementioned algorithms to discern among different elements appearing in the captured scene.

To capture all this information, hyperspectral sensors build a three-dimensional (3D) data structure –hereafter hyperspectral cube or HSI cube– in which the first two

dimensions present the spatial information, while the third one does likewise with the spectral information. This data structure is shown in Figure 2-5. On the left, a simplification of the structure of the HSI cube is shown, representing the information of a few wavelengths; on the right, two different images are presented: at the top, a representation of the spectral signature associated to a specific pixel, with a length equal to the number of spectral bands of the HSI cube; at the bottom, a gray scale image of the captured scene, representing the result that can be obtained when selecting a single spectral band, displaying the spatial information captured by the sensor at that particular wavelength.

Figure 2-5: Simplified structure of a hyperspectral cube, showing the expected single band representation at a given wavelength and the spectral signature associated to a single pixel. Based on a figure from [Halicek'19].

As a result, HSI is a non-invasive technique that enables rapid acquisition and identification of the captured scene. As mentioned before, it was initially developed for remote sensing [Govender'07, Meer'12], but its potential has caused an increase of its applicability, spreading to many different research fields, such as agriculture [Dale'13], forensics [Edelman'12], archeology [Liang'12], food safety [Wu'13], defense and security [Yuen'10], drug identification [Carvalho Rocha'11] and medical applications [Lu'14, Calin'14]. Specifically, this technology has had a wide impact in this last mentioned field, since it has been proven that the interaction of the tissues with the electromagnetic radiation generates useful information to assist in diagnosis processes [Lu'14]. In addition, HSI is a non-invasive, non-ionizing and non-contact technique, which are very interesting features that make HSI a competitive alternative to already existing diagnosis technologies. This technology has been already applied in different areas to assist in medical diagnosis, namely histopathological tissue analysis [Lu'13, Ortega'18], intestinal ischemia diagnosis [Akbari'10], estimation of cholesterol levels [Milanic'15], blood vessel visualization enhancement [Bjorgan'15] and, more recently, cancer detection [Kiyotoki'13, Regeling'16a, Ortega'19].

2.4.2 Hyperspectral Imaging (HSI) for cancer analysis

As aforementioned, cancer detection is one of the medical fields in which HSI is gaining momentum. During the last years, the application of this technology to assist in cancer diagnosis has gradually increased, being used with different types of tumors with promising results. This section provides a review of the state-of-the-art research in this field. A more detailed analysis of this topic can be found in [Halicek'19].

As stated in [Halicek'19], the research devoted to apply HSI to cancer detection can be summarized following a hierarchical structure, like the one shown in Figure 2-6. This structure presents three different levels: first, the research is categorized depending on the organ in which the tumor is located; then, the type of tissue samples and nature of the experiment; and, finally, the nature of the subjects from which the image has been taken. Specifically, the research lines presented in this section are first classified as *gastrointestinal*, *breast*, *head and neck* and *brain*, depending on the type of cancer; then, each of them is classified in *ex-vivo*, if the samples that are captured by the HSI system have been previously extracted from the host, or *in-vivo*, if not; and, finally, *human* or *animal*, depending on the nature of the subject.

Figure 2-6: Taxonomy of the main research fields of HSI for cancer detection. Based on a figure from [Halicek'19].

Gastrointestinal cancer, most frequently caused by dietary reasons or infection with *helicobacter pylori*, is the second leading cause of cancer death globally, presents a 5-year survival of only 10% for more than half of the cases diagnosed as locally advanced or metastatic and its only curative therapy is still surgical resection [Orditura'14]. Surgical resections of gastric carcinomas have also evolved, and nowadays the most common technique is laparoscopy, which is guided by an endoscope and, therefore, it is minimally invasive. However, a drawback of this technique is that surgeons lose a tactile feedback that they often require [Ottermo'06]. To overcome this limitation, HSI has been proposed as a solution to provide more accurate digestive tract tumor resections.

The first studies to this regard were performed in human ex-vivo tissues, with an HSI system that captured information in 239 spectral bands and succeeded in discerning the boundaries between normal and tumor tissue validating the results with histopathological analysis [Akbari'11]. These studies have continued during the last years; one of the most recent ones presents a laparoscopic HSI system that uses collectively two cameras to distinguish among healthy colorectal mucosa, normal fatty tissue and adenocarcinoma to give extra information to the surgeons so as to compensate the aforementioned loss of tactile feedback [Baltussen'19].

With respect to in-vivo experiments, HSI endoscopic systems have been primarily used to assess colorectal tumors. One of the most relevant studies in this domain used a flexible hyperspectral colonoscopy system capable of capturing information at 27 different wavelengths and discriminating between normal colonic mucosa and colorectal tumors in human patients [Han'16], thus proving the fitness of HSI for delimiting the disease region in-vivo and providing a sensitivity and a specificity up to 96% and 91%, respectively, being the former the capability of correctly identifying tumor tissue as cancerous –i.e., true positive–, and the latter the ability to correctly identify a healthy tissue as non-cancerous –i.e., true negative.

Breast cancer has traditionally been treated with a mastectomy, a surgical procedure in which the whole breast is removed. However, in early-stage breast cancer, the preferred method is a lumpectomy, which is a partial removal of the breast that is afterwards combined with radiation therapy with the objective of conserving the breast [Hwang'13]. However, the success of this technique heavily relies on completely removing the tumor mass; as an example, some studies report that around 20% of patients undergoing lumpectomies need additional surgeries [Jacobs'08], which in turn implies extra intraoperative biopsies and pathologist consultations to guide surgeons; to that end, HSI presents itself as a good alternative.

The objective of many human ex-vivo breast cancer studies is to delineate the Regions Of Interest (ROIs) in the resected samples to discern between normal and tumor tissue, as shown in two different studies from 2013 [Kim'13, Pourreza-Shahri'13]. The first one achieved similar results than those of a manual segmentation while performing automatic ROI identification using contrast and texture information [Kim'13]; as for the second, it achieved sensitivity and specificity rates of 98% and 99%, respectively, when combining supervised classification techniques with feature extraction and dimensional reduction algorithms to automatically distinguish between normal and cancerous tissue, thus proving the potential of this technology to aid in human breast cancer diagnosis [Pourreza-Shahri'13].

Additionally, some experiments have set the basis for applying this technology in-vivo, although they have been carried out using animals. One of the most promising ones used HSI to capture information at 34 different wavelengths while performing surgery on 60 rats previously induced with breast cancer [Panasyuk'07], and succeeded in accurately identifying blood vessels, muscle, connective tissue and tumor. Additionally, histopathological analysis ensured sensitivity and specificity rates of 89% and 94%, respectively, when detecting residual tumor.

Head and neck cancer is the sixth most common cancer around the world [Vigneswaran'14]. It includes cancers originating in the upper aerodigestive tract, namely oral and nasal cavities, pharynx and larynx. Around 90% of them are squamous cell carcinoma, which usually expresses itself at an advanced stage of the disease, and the main treatment remains surgical resection [Yao'07]. As in many types of cancer, the single largest predictor of patient outcomes is the complete removal of the carcinoma from the wound bed; otherwise, the probabilities of disease recurrence and additional surgeries are greatly increased [Molony'17]. Even with pathologist consultations during surgery to guarantee a complete extraction, the literature reports that up to 20% of patients relapse despite having negative margins in the operation –i.e., removing the entire tumor–, which in turn justifies the estimated five-year survival rate of 40-60% with treatment [Hernández-Vila'16]. For those reasons, HSI can be a good candidate to provide extra information to guide the surgeon during the operation.

Ex-vivo experiments applying HSI to head and neck tumors are one of the most exploited topics in the literature. In 2017, several works were published with the objective of discerning cancerous and non-cancerous tissue: Fei et al. [Fei'17] achieved promising results in identifying the boundaries between normal and tumor tissue from ex-vivo samples and comparing those results against the histopathological ones; Lu et al. [Lu'17] addressed a comparison of different machine learning techniques with both intra- and inter-patient classifications; and Halicek et al. [Halicek'17] used deep learning techniques to distinguish between normal and tumor tissue, comparing the results with those obtained with traditional machine learning techniques.

HSI literature also presents works with in-vivo samples from human subjects. In 2011, Jayanthi et al. [Jayanthi'11] studied early diagnosis of malignant changes in the oral cavity, capturing information from 40 bands and obtaining sensitivity and specificity rates over 95%. In 2016, Regeling et al. [Regeling'16b] employed a flexible endoscope coupled with an HSI system to capture in-vivo information at 30 wavelengths to identify laryngeal cancer, achieving an overall accuracy of 88%.

Brain cancer and, more specifically, malignant gliomas cause between 2% and 3% of cancer deaths around the world, according to [Halicek'19]. Major treatment options include surgery, chemotherapy and radiotherapy [Linck'06]. However, one of the main problems of brain tumors is that they do not present clear boundaries, but diffuse ones, since they infiltrate into the surrounding healthy brain tissue. As a result, surgeons are often unable to accurately discriminate between tumor and healthy tissue, and they must rely too much on the information coming from their naked eyes. Consequently, it is very frequent that, during neurosurgeries, too much healthy brain tissue is removed as a safeguard –the so-called *safety margin*–, or, on the contrary, that some tumor tissue remains without being noticed –hereafter, *residual tumor*–. On the one hand, various studies in the literature have proven that this residual tumor is the most frequent cause of recurrence and a major cause of mortality and morbidity [Sanai'08]; on the other hand, over-resection of tumor tissue has been proven to provoke permanent neurological damages, jeopardizing patients' quality of life [Stummer'11]. As a result, this type of tumor requires a delicate balance between completely removing the tumor

while minimizing as much as possible the resection of healthy brain tissue to maximize patients' quality of life. To achieve this extremely accurate margin delineation, several guidance image tools are commonly used to assist surgeons:

- **Intraoperative neuronavigation:** Its main drawback is that it is affected by the so-called *brain shifting* phenomenon, which represents the brain deformation produced after the skull is opened and the brain is exposed [Gerard'17].
- **Intraoperative Magnetic Resonance Imaging (MRI) (iMRI):** This technology provides good assistance to surgeons, but it presents relevant drawbacks: first, it considerably extends the duration of the surgery in a range of 20 to 75 minutes per image taken [Chowdhury'18]; subsequently, the number of images that can be taken during a surgery is limited; finally, it requires special operating rooms.
- **Intraoperative Ultrasound (iUS):** This technology presents many advantages, since it is not expensive, can be used in real-time and is not affected by the brain shifting phenomenon [Chacko'03, Young'15]. However, its use can cause the resection of normal tissue, which is precisely one of the drawbacks to avoid.
- **5-Aminolevulinic Acid (5-ALA):** Fluorescent tumor markers have the advantage of being able to identify the tumor boundaries, but it can only be used in high-grade tumors and, even more importantly, it is invasive and produces relevant side effects, as convulsions or epidural hematoma [Stummer'06].

As can be observed, all these technologies can be useful in guiding neurosurgeons, but all of them present, at least, an important disadvantage. Hence, HSI can be a potential solution, as it is a non-invasive, non-ionizing and label-free image modality. For those reasons, some relevant works in the recent literature have explored the applicability of this technology to in-vivo human brain cancer to accurately identify tumor margins [Szolna'16, Salvador'17, Fabelo'18a]. All these studies are a result of a European project named HypErspectraL Imaging Cancer Detection (HELICoiD), whose objective was to generalize a methodology to distinguish between normal and cancerous tissues in real-time during neurosurgical procedures [Fabelo'16b]. By the end of this project, an intraoperative demonstrator was designed, built and tested to acquire intraoperative HSI and process them in situ during the neurosurgery. The images were processed in near real-time applying their own classification algorithm, developed during the project and composed of a supervised classification phase coupled with an unsupervised segmentation method through a majority voting procedure. The results were evaluated with an intra-patient cross-validation method, obtaining specificity and sensitivity rates over 98% [Fabelo'18b]. Subsequent studies proved the potential of this methodology in identifying different types of brain tumors, not only high-grade gliomas [Fabelo'18a]. Also in the context of this project, several studies have been published comparing both different processing methods, such as machine learning and deep learning [Madroñal'17b, Fabelo'19a], and different hardware accelerators, namely many-core and GPUs [Lazcano'19b]. Additionally, another relevant outcome of this project was the creation of the first open-access HSI database for in-vivo human brain tumor images [Fabelo'19b].

2.4.3 Hyperspectral Imaging (HSI) processing methods

As mentioned before, the application of HSI to identify objects appearing in the captured scene has been extensively studied in the literature. Accordingly, the topic of how to perform this identification has also been thoroughly analyzed in the literature [Li'14a]. As this technology was initially developed for remote sensing applications, the first algorithms devised to address this classification are linked to this field [Fauvel'12], but the continuous expansion of this technology to other fields has reopened, in turn, this discussion, revisiting the existing algorithms to classify different scenes, apart from those coming from the remote sensing field. The remaining of this section is going to review the main methods for HSI classification, together with the most used algorithms to implement each one. These methods are *supervised classification*, *unsupervised classification* and *deep learning*. As established in [Halicek'19], these three methods suppose each pixel to be either pure or a combination of pure ones, that is, that each pixel can be assigned to a certain material or to a combination of several materials appearing in the scene based on its spectral information [Chang'03].

Supervised classification is based on building a model from a set of spectral signatures with known class labels –i.e., the material each spectral signature defines is known beforehand– and using this trained model to assign new labels to the unknown spectral signatures of the image. To complete this training, a set of already labeled spectral signatures must be available, and the size of this set must be such that the set can be considered as sufficiently representative, so it can be used for generalization purposes. With respect to the algorithms that are most widely used in the literature for performing this classification, they are usually statistical machine learning or regression-based methods that rely on techniques such as decision trees, linear discriminant analysis, Artificial Neural Networks (ANNs) [Benediktsson'05], Random Forests (RFs) [Chan'08] and kernel-based methods [Camps-Valls'05]. Within this last category, Support Vector Machine (SVM) classifier is one of the most widely applied algorithms in HSI and the preferred one among researchers, since it provides a good tradeoff between performance and number of training samples required. As stated in [Lu'14], the rationale behind this wide applicability is manifold: strong theoretical foundations, low sensitivity to the dimensionality, good performance with a limited number of training samples and good generalization capabilities. Considering HSI medical applications, many works in the literature also demonstrate the reliability of SVM with a wide variety of medical images [Wang'16, Zhang'16b].

Unsupervised classification divides the image into a set of groups –hereafter, *clusters*– that share very similar spectral information. These groups are created by iteratively refining the computation of the *cluster centroid*, which represents the spectrum associated to a material in the captured scene, and associating the rest of the pixels to one of the centroids [Dey'10]. The main difference with supervised classification methods is that unsupervised classifiers do not require either a training process or previously labeled samples; hence, these methods can only provide relative clustering information, without any identification of the material to which each pixel belongs. Although it cannot provide identifying information, it can be useful to

pinpoint the boundaries of the different spectral regions appearing in the scene. As established in [Li'14a], the most common unsupervised classification algorithms applied in HSI applications are K-Means [Jain'10] and Iterative Self-Organizing Data Analysis (ISODATA) [Ball'65, El Rahman'15]. More specifically, some recent works in medical HSI show the usage of unsupervised clustering as a part of further classification algorithms [Masood'06, Regeling'16b].

Deep learning techniques have been increasingly applied in remote sensing applications for pixel-wise classification, target detection and image processing [Zhang'16a]. As established in [Halicek'19], deep learning is based on generating predictive models conformed by a number of processing layers with the so-called *neurons* that are activated to discriminate among different representations of data and multiple levels of abstraction. These methods are able to extract features from very large datasets by following an iterative and error backpropagation approach [LeCun'15]. This method presents an important advantage with respect to traditional machine learning techniques: they can learn novel mathematical representations of the input thanks to non-linear modules that modify the data representation from the original level to a slightly more abstract level, in which more complex functions can be learned [LeCun'15]. There are many deep learning algorithms that have been applied to HSI in the literature, namely Deep Belief Networks (DBNs) [Hinton'09] and Convolutional Neural Networks (CNNs) [LeCun'98], and which offer improved results than those obtained with traditional SVM-based classifiers [Li'14b, Makantasis'15]. More specifically, in the medical field, deep learning is an emerging and powerful technique, which has started to be exploited in the recent years [Ravi'16, Tabar'16, Korbar'17]. However, the full exploitation of deep learning techniques for medical HSI is yet to come for two main challenges: first, it requires very large amounts of training samples, which by itself is a significant challenge, since nowadays there is not a large amount of such databases available; and, secondly, the inter-patient spectral variability forces the size of those databases to be much larger than those non-medical ones so as to guarantee generalization capabilities.

3

METHODOLOGY SPECIFICATION AND VALIDATION

Chapter 1 presented the motivation and objectives of the work described in this document, while Chapter 2 introduced all the concepts, methodologies and frameworks that play a role in the ideas, proposals and solutions presented in this work, together with a brief analysis of the state of the art related to those methodologies and frameworks. This chapter serves as a bridge between those chapters and the ones devoted to describe in detail the contributions of this PhD work; its main objective is to contextualize the solution proposed in this document.

To do so, the remaining of the chapter is organized as follows: first, Section 3.1 will describe in detail what is the problem that has been identified in the state of the art and what is the solution devised in this document to address the mentioned problem; afterwards, Section 3.2 will outline the validation methodology that is going to be followed to evaluate the solution both functionally and in terms of performance.

3.1 Methodology specification

As described in Chapter 1, the trend in applications during the last few years can be defined as an increment both in their complexity and in their performance requirements, which in turn has been translated into an increase in the complexity of the parallel architectures specifically devised for trying to reach these performance requirements by augmenting the computational power and the physical PEs available.

To explore how this problem has been addressed in the recent literature, Section 2.2 presented a brief overview of the state of the art related to application acceleration, reviewing some of the most extended methodologies to parallelize applications. Table 2-2 –repeated in Table 3-1 to ease the reading of the document– closed this section, providing a comparison of the different reviewed methodologies. This comparison was performed in terms of the types of parallelism each methodology exploited and the level of abstraction it provided to developers. As concluded in Subsection 2.2.5, this table shows that, in general, there is a compromise between these two factors: increasing the level of abstraction to enhance developers' productivity comes at the cost of sacrificing some of the types of parallelism that can be exploited.

This table also sheds light upon the motivation of this work: as can be easily inferred from the table, if dataflow-based design methodologies are combined with automatic parallelization techniques, the resulting methodology would be able to exploit the most common types of parallelism while still maintaining the high level of abstraction, which is crucial to promote developers' productivity when dealing with ever increasing complex applications and architectures.

Table 3-1: Summary of the different parallelization methodologies and their evaluation (repeated from page 26).

	Parallelism					Abstraction level
	Instruction level	Data level	Loop level	Task level	Pipeline level	
Programming APIs	✓	✓	✓	✓	✓	↓
Dataflow-based design	✗	✓	✗	✓	✓	
Automatic parallelization	✓	✓	✓	✗	✗	

As shown in Section 2.2, there are many tools and frameworks that have been developed in the last years to address either dataflow-based design or automatic parallelization. Consequently, to build such a framework as the one proposed in this document, it does not seem wise to start from scratch, but to take advantage of all the knowledge already existing in those fields to build this enhanced methodology. To do so, first the requirements this methodology must fulfill are going to be enumerated; afterwards, they will be used to compare the frameworks and tools presented in Subsection 2.2.3 and Subsection 2.2.4. The objective of this study is to choose a framework from each area –dataflow-based design and automatic parallelization– to serve as a starting point to begin building this enhanced methodology.

As advanced in Chapter 1, there are some requirements this methodology must fulfill, which come from the analysis of the features demanded by applications nowadays. As already stated, the increase in the complexity of both applications and architectures has created a need for: (i) exploiting the different types of parallelism that can appear within an application, so that it can be efficiently deployed over heterogeneous architectures; and (ii) isolating the modeling of the application from that of the architecture so as to simplify this heterogeneous deployment. However, the ever-tightening requirements mentioned before may not always be related to performance, but also to self-awareness and self-reconfiguration; for instance, it is quite common for the response of an application to depend on some external events that occur at runtime; hence, the application also needs to (i) be aware of those changes at runtime, and (ii) reconfigure itself depending on that event [Palumbo'19a]. This kind of behavior makes compulsory the existence, for instance, of some kind of runtime manager to deal with this dynamic awareness and reconfigurability.

These requirements are derived from applications such as the use-case proposed in this document –applying HSI technology to locate human brain tumor boundaries during surgeries–, since it involves computationally complex algorithms –i.e., machine learning techniques– that must be deployed onto heterogeneous architectures to be

parallelized; additionally, this kind of application also involves dynamic reactivity, since it should adapt itself to runtime changes, as a change in the image resolution.

Consequently, the requirements this methodology must fulfill are the following:

- To be able to exploit different types of parallelism.
- To keep a high level of abstraction, so developers can increase their productivity.
- To make use of Y-chart design methodology to isolate the application from the architecture, which helps in maintaining the level of abstraction.
- To be aware of changes at runtime, so applications can reconfigure themselves if such changes appear.

These are the generic requirements the methodology demands, but, to choose frameworks from the state of the art to build the basis of this methodology, some other requirements, more specific to each domain, appear:

- In general lines, both frameworks must be open-source. This is a very important requirement, since, in order to build this methodology with previous works as a basis, the source code must be available.
- As already stated, those frameworks should be reactive to changes in runtime.
- With respect to dataflow-based design, two specific requirements appear:
 - In order for such a system to work, it is important that some guarantees can be offered to the user. Specifically, when describing MoC properties in Subsection 2.1.1, one that was especially interesting was *decidability*, since it implies that applications are deadlock-free by construction.
 - When describing MoC properties, an interesting property is *parallelization*. The level of automation with which the frameworks presented in Subsection 2.2.3 exploit the different types of parallelization varies considerably. For that reason, another important feature is being able to exploit the types of parallelism associated to dataflow-based design –data-, task- and pipeline-level– in an automatic or semi-automatic fashion.
- As mentioned before, one of the most important requirements is that the new framework must be reactive to changes at runtime. Regarding automatic parallelization, this means that, if possible, the changes applied by these frameworks to accelerate applications must be applied at runtime. As a result of this, another two specific requirements arise:
 - To be reactive to changes at runtime and reconfigure applications and optimizations accordingly, it is desirable that the automatic parallelization framework has full runtime management, that is, that a runtime manager is in charge of applying these optimizations to be able to consider runtime information –e.g., current values of loop bounds, parameters, etc.

- Additionally, to grant as much control power as possible to this runtime manager, it is desirable that the automatic parallelization framework has some control over the selected optimization so as to have the possibility of choosing or changing this optimization depending on the specific context.

With these key features in mind, a comparison of the frameworks presented in Subsection 2.2.3 and Subsection 2.2.4 has been carried out. Table 3-2 gathers the comparison of the dataflow-based design frameworks, while Table 3-3 does the same with the automatic parallelization ones. In both tables, these features appear marked as fulfilled by the specific framework (✓) or not (✗). For those in which a blank space is left, it means that no conclusive information has been found to that respect.

Table 3-2: Summary of the different dataflow-based design frameworks and their evaluation through the characteristics defined to compare them. A blank space indicates that no clear information has been found to that respect.

Frameworks	Open source	Runtime reactivity	Deadlock freeness	Semi-automated parallelization
SynDEx	✗		✓	✓
MAPS	✗	✗	✗	
ORCC	✓	✓	✗	✗
PREESM + SPiDER	✓	✓	✓	✓
PN - COMPAAN TOOL	✗	✗	✗	

On the one hand, as shown in Table 3-2, only two of the frameworks reviewed in the literature –ORCC and PREESM with its dynamic runtime manager, SPiDER– are open-source and present the mechanisms needed to be aware and reactive to changes at runtime. However, applications modeled with ORCC cannot be guaranteed to be deadlock-free by construction, since the MoC upon which ORCC is built, DPN, is not *decidable*. Additionally, this framework neither includes semi-automated parallelization: although applications represented with ORCC can be parallelized, this process must be performed manually by the developer. On the contrary, PREESM is based on a *decidable* MoC, PiSDF, so its applications are guaranteed to be deadlock-free by construction, and it presents a semi-automated mechanism to exploit data-level parallelism, since users can easily replicate actors as many times as they require without having to manually do it, just modifying some parameters in the application. Finally, another aspect that should be considered is that PREESM counts with a very active developer community, while ORCC has been recently marked as deprecated. For all those reasons, PREESM –and its dynamic counterpart, SPiDER– is the dataflow-based design framework selected to be one of the cornerstones of the new methodology proposed in this document.

On the other hand, as depicted in Table 3-3, although there are many tools providing automatic parallelization techniques, most of them do not provide runtime support, that is, they can only apply these optimization techniques at compile-time. Only three open-source frameworks support the application of these techniques at

Table 3-3: Summary of the different automatic parallelization techniques and their evaluation through the characteristics defined to compare them. A blank space indicates that no clear information has been found to that respect.

Frameworks	Open source	Runtime reactivity	Runtime manager	Optimization control
Pluto	✓	✗	✗	✓
Polly	✓	✗	✗	✓
GRAPHITE	✓	✗	✗	✗
CHILL	✓	✗	✗	✓
AlphaZ	✓	✗	✗	✓
R-Stream [®]	✗	✗	✗	
IBM XL	✗	✗	✗	
GeCoS	✓	✗	✗	✗
PPCG	✓	✗	✗	✗
PENCIL	✓	✗	✗	✗
TIRAMISU	✓	✗	✗	✓
SPolly	✓	✓	✗	✓
ExanaDBT	✓	✓	✓	✗
Apollo	✓	✓	✓	✓

runtime: SPolly, ExanaDBT and Apollo. So as to choose the most suitable framework for the purpose of this PhD work among these tools, the other key features are going to be taken into consideration. As aforementioned, in order to help in fulfilling the requirements the new methodology demands, it is important for the automatic parallelization framework to have a full runtime manager, which is in charge of (i) handling the generation and execution of the different optimizations depending on the specific context, and (ii) selecting somehow the optimization to apply among all the possible ones –i.e., that it has control over which optimization is going to be applied.

In the case of SPolly, this framework fulfills the latter, but not the former, since several different versions can be created, but they must be created at compile-time; at runtime, the only task is instantiating them with the specific values gathered by a profiling stage. This does not reach the level of flexibility required, so this framework cannot be considered the most suitable candidate. As for ExanaDBT, the opposite situation appears: this framework provides a runtime manager to handle the execution, but it has little control over which optimizations are going to be applied; thus, this framework is also removed from the list of candidates. Finally, in the case of Apollo, from Table 3-3 it can be seen that both requirements are fulfilled: Apollo provides a runtime manager to handle the execution flow of the application and also to choose the optimization to apply at each time. Hence, Apollo has been selected as the other cornerstone upon which the new methodology will be built.

To sum up, the methodology proposed in this PhD is built upon two cornerstones: PREESM –and SPiDER– from the dataflow domain, and Apollo from the automatic parallelization one –or *polyhedral domain*–. These frameworks will be fully described in Chapter 4, where the knowledge about them will be completed and extended.

3.2 Validation methodology

As advanced in Section 2.3, when designing a new methodology, the validation must prove two crucial features: first, that the new methodology functionally works, if possible proving its correctness through a set of applications widely used in the related literature, so it can be compared with possible competitors; secondly, that the new methodology provides the advantages for which it was built in the first place, if possible with a real use-case that could use this methodology to exploit these advantages.

Hence, the validation process followed in this document is composed of two steps: first, a functional validation through a set of benchmarks; afterwards, a real use-case validation through a real-life application modeled using the proposed methodology.

These two steps complement each other in such a way that, if one of them is not present, the validation is not complete. On the one hand, if the methodology is validated only through benchmarks, the initial hypothesis for which the new methodology has been developed –i.e., providing developers with more resources to exploit the parallelism of current applications– would not be proved, since benchmarks are really useful for comparison reasons but they usually represent isolated, specific problems that cannot be considered as applications by themselves. On the other hand, if the methodology is only validated through a real-life application, comparing the obtained results against those provided by state-of-the-art competitors becomes highly difficult, since normally it is up to the developer to implement the application following different methodologies to compare them.

The rest of this section justifies the choices made around this topic, presenting the validation methodology followed to evaluate the solution proposed in this document. In that line, Section 2.3 presented a brief overview of the benchmarks most widely used either in the dataflow domain or in the automatic parallelization one.

On the one hand, with respect to the dataflow domain, as advanced in Subsection 2.3.1, although there are some works in the literature trying to provide a unified benchmark suite for this domain, the versatility in the characteristics of the different MoCs makes impossible to devise an unified benchmark suite, so these works have had little impact in the community. The most common practice is, therefore, for each framework provider to develop a small set of applications that are used by themselves as benchmarks, so they can be used to validate new functionalities. PREESM, which is the dataflow-based design framework chosen in this work as a basis, has a very active developer community, and they have developed their own set of applications, which are available online¹. For those reasons, in this document some of those applications will be used as a validation from the dataflow point of view.

On the other hand, Subsection 2.3.2 reviewed the different benchmark suites most widely used to validate automatic parallelization methodologies and frameworks. In this case, there is a benchmark suite that clearly stands out among all the others,

¹<https://github.com/preesm/preesm-apps>

which is **PolyBench**. This suite has been widely used in the automatic parallelization community, and more specifically, in the polyhedral domain community, since almost all the tools presented in Subsection 2.2.4 make use of this suite to validate their implementations. As a result, this benchmark suite has been selected to validate the proposed methodology from the automatic parallelization point of view.

However, although this benchmark suite is, without a doubt, the most used one, it lacks some key features that would be interesting for the methodology proposed in this document. As already stated, the automatic parallelization framework upon which this methodology is built is Apollo. This framework has been chosen because it is one of the few existing in the state of the art that successfully extends the application of the polyhedral model to the runtime domain, hence widely increasing its applicability and its parallelization potential. But, because of that, PolyBench is not enough to fully validate changes in this tool, since this suite is specifically developed for static frameworks. For that reason, Apollo was originally validated with its own benchmark suite, which will be referred to as **ApolloBench** hereafter. This benchmark suite was built, in turn, from a collection of benchmark suites, and the chosen kernels were specifically selected to highlight the capabilities of this tool. The benchmark suites from which these kernels have been selected are: **Scimark**, **Rodinia**, **SPARK00** and **Parboil**, all of them previously described in Subsection 2.3.2. As stated in [Caamaño'16], these kernels were selected identifying some desired characteristics that make impossible their parallelization at compile-time, such as the usage of pointers, indirections, unpredictable loop bounds and unpredictable scalar values. Consequently, **ApolloBench** will also be used to validate the proposed methodology from the automatic parallelization point of view, together with **PolyBench**.

Consequently, the functional validation of the proposed methodology is carried out through three benchmark suites: PREESM applications, PolyBench and ApolloBench.

From the point of view of the real use-case validation, Section 2.4 introduced the use-case selected in this work, which is an application to process hyperspectral images in real- or near real-time to provide an accurate classification of the scene under observation to neurosurgeons to assist them in locating the boundaries of tumor tissue in human brain cancer surgeries. This application is chosen for four main reasons:

- As already mentioned, this application presents a strict performance requirement, since the result must be provided to the surgeon as soon as possible.
- The volume of information to be processed is very large, as hyperspectral images capture information across hundreds of bands in the electromagnetic spectrum.
- The algorithms usually applied to process HSI are very complex –machine learning, deep learning– and require parallelization to be executed in real-time.
- As shown in Subsection 2.4.2, the domain of this application –applying HSI technology to medical applications and, more specifically, to cancer detection– has gained momentum in the last few years, as it can cause an important impact on patient's quality of life and prognosis.

This application completes the two-phase validation methodology proposed to evaluate the solution presented in this work. Chapter 7 will provide an extensive description of all the elements involved in this validation methodology, describing in detail both the kernels that have been selected from the benchmark suites to address the functional validation and the specifics of the application that is going to be used to process these hyperspectral images and complete the real use-case validation.

4

TECHNOLOGY FRAMEWORK

Chapter 3 justified the choice of the frameworks that will be used as a starting point for this work: Apollo from the polyhedral domain, and PREESM and SPiDER from the dataflow one. In this chapter, these tools are going to be described in detail to have a deep understanding of their functioning before any modification is addressed.

This chapter also serves to discern between what was already done and the contributions of this work to these frameworks. The rest of the chapter is organized as follows: first, Section 4.1 introduces Apollo and its associated ideas; likewise, Section 4.2 outlines both PREESM and SPiDER, PREESM's runtime resource manager.

4.1 Polyhedral domain: Apollo

As stated in Subsection 2.2.4, Apollo [Caamaño'16] is a hybrid compilation framework that effectively extends the polyhedral model to reach any kind of loop nests, even if they include indirections, unpredictable memory accesses through pointers and/or unpredictable loop bounds, as long as they are compliant with the polyhedral model at runtime. Its strategy to make the polyhedral model applicable for codes that cannot be analyzed statically is based on speculation coupled with runtime verification.

Before getting into the details of the tool, first some concepts must be described, as they are the cornerstones upon which the foundations of Apollo lie. These concepts are introduced in Subsection 4.1.1; then, Apollo is detailed in Subsection 4.1.2.

4.1.1 Previous concepts

As aforementioned, Apollo is built upon two main cornerstones: the polyhedral model and TLS. These concepts were briefly introduced in Subsection 2.2.4, but they need to be described in detail so as to simplify the understanding of Apollo, and how it works.

Polyhedral model

As already said, the polyhedral model [Bondhugula'08, Feautrier'11] is a powerful mathematical framework to analyze and optimize loop nests. This model represents individual statement iterations as integer points inside a polyhedron, which eases the analysis and transformation of loops. To comply with this model, a loop must fulfill some requirements: (i) each loop has to iterate according to a unique index variable, whose bounds are affine expressions of the enclosing loop indices –i.e., bounds expressed as a linear function of the loop indices, plus a constant–; and (ii) the memory instructions are limited to access simple scalar variables or multi-dimensional array elements referenced using affine expressions on the enclosing loop indices. Such loop nests are analyzed with respect to data dependences that occur among the statements and across iterations. Hence, advanced optimizing transformations are proven to be semantically correct by preserving the dependences of the original program.

Thus, transformations represent reorderings of the iterations and are defined as scheduling matrices, which are equivalent to geometrically transforming a polyhedron into an equivalent form. Representing loop nests as polyhedra enables one to reason about the valid transformations that can be performed.

To go through the main concepts of the polyhedral model, the loop depicted in Figure 4-1a is going to be used as a toy example [Yuki'13].

```
/* original source code */
for(i=0; i<=N; i++){
  for(j=0; j<N-i; j++)
    S(i, j);
  for(j=N-i+1; j<=N; j++)
    T(i, j);
}
```

(a) Example of a loop nest

(b) Iteration domains representation

Figure 4-1: Example of a loop nest and polyhedral representation of its iteration domains for $N = 3$. The dotted red arrow indicates the original execution order.

In this model, a loop is composed by a number of statements. Each statement has its own *iteration domain* –defined by the loops surrounding the statement– and its *schedule* –determining the execution order of the statement instances–. As can be observed, the loop showed in Figure 4-1a has two abstract statements, S and T . Their respective iteration domains, D_S and D_T , are presented in Equation 4-1 and Equation 4-2, respectively. They represent the values that the iterators of the loops surrounding each statement can take. The optimizations that could be applied to this loop are linear transformations of the iteration domains, represented geometrically as polyhedra, as can be observed in Figure 4-1b, where the dotted red arrow shows the original execution order, which corresponds to the lexicographic order.

$$D_S(N) = \{i, j \mid 0 \leq i < N \wedge 0 \leq j < N - i\} \quad (4-1)$$

$$D_T(N) = \{i, j \mid 0 \leq i \leq N \wedge N - i < j \leq N\} \quad (4-2)$$

As already stated, the objective of transforming a loop nest is to find a new execution order in which either some internal parallelism is exposed or the locality of memory accesses is improved, but ensuring that no dependence is violated. Considering the code snippet presented in Figure 4-1a, and assuming that abstract statements S and T are independent from each other, one possible transformation would be the one shown in Figure 4-2. The original loop is now transformed into the code shown in Figure 4-2a, while Figure 4-2b presents the associated iteration domain. As can be observed, thanks to this transformation one of the innermost for loops has completely disappeared, which is a transformation known as *loop fusion*.

```
/* original source code */
for(i=0; i<N; i++){
    for(j=0; j<N-i; j++){
        S(i, j);
        T(N-i, N-j);
    }
}
```

(a) Transformed loop nest

(b) Representation of the new iteration domain

Figure 4-2: Loop transformed and polyhedral representation of its new iteration domain.

Some of the most relevant transformations that can be applied to a loop have been fully detailed in [Bacon'94]. The remaining of this section provides a brief summary:

- **Loop fusion:** This transformation focuses on combining the bodies of two loops iterating over the same iteration domain. By doing so, one can reduce loop overhead, increase instruction parallelism, improve register, vector, data cache or page locality, and promote the load balance of parallel loops.
- **Loop fission:** This transformation is the opposite of the previous one, so it breaks a loop into several ones. All the new loops share the same iteration domain, but only contain a small subset of the statements included in the original loop. This transformation is usually applied to create loops with fewer dependences, to improve instruction locality and cache –because of shorter loop bodies–, and to reduce memory requirements by iterating over less arrays.

- **Loop interchange:** This transformation exchanges the position of two loops, generally permuting one of the outer loops with the innermost. It can improve performance in several ways, namely: enable vectorization by interchanging an outer, independent loop with an inner, dependent loop; promote vectorization by placing the independent loop with the largest range in the innermost position; and improve parallel performance by moving an independent loop outward in a loop nest to increase the granularity of each iteration so as to reduce the number of barrier synchronizations. However, these benefits could cancel each other out, so it must be taken into consideration when applying this transformation.
- **Loop reversal:** It modifies the direction of the iteration domain, thus changing the dependence vectors, and it is usually combined with other reordering transformations. It can reduce loop overhead or eliminate temporary variables. Additionally, it can also enable further optimizations, such as loop interchange, in such a way that it is possible to apply it after reversal, but not before.
- **Loop skewing:** Usually coupled with loop interchange, this transformation was devised to face the issue of wavefront computations, which received that name due to the wave-like form in which the updates to the array propagate across the iteration domain. It is performed by multiplying the outer loop index by a *skew factor*, f , and adding it to the bounds of the inner iteration variable, subtracting the same value every time this variable is used inside the loop [Bacon'94]. This transformation aligns the diagonal wavefronts of the original loop nest, allowing the parallelization of the inner iterations, not parallelizable before.
- **Loop tiling:** This transformation is based on splitting a loop with a large number of iterations into several *tiles* iterating over smaller iteration domains. As a result, tiles work on smaller datasets, thus improving data locality. Additionally, when combined with other transformations, further optimizations can be achieved. For instance, if combined with skewing, coarse-grain parallelism can be exposed, since it allows the execution of several tiles in parallel, following a wavefront fashion. Likewise, fine-grain parallelism within a tile can be exposed by pipelining or vectorizing the execution of that tile.
- **Loop unrolling:** This transformation aims at replicating a loop as many times as stated by the *unrolling factor*, u , iterating over it by step u . It may bring forward some relevant optimizations, namely reducing the loop overhead, increasing instruction-level parallelism and promoting data cache and register locality.

Although the example shown in Figure 4-2 has served as a toy example to illustrate how the polyhedral model works, this example is not enough to fully understand the optimization possibilities this model can offer. For that reason, to conclude its description, an example of a real transformation is going to be provided. To that end, Figure 4-3 provides both the original loop kernel without being transformed, and the resulting loop kernel after applying an optimizing transformation, together with the iteration domains associated to each of those loops.

As can be observed, the original loop cannot be parallelized in any way: each iteration of each loop depends on previous iterations, making impossible the parallelization of any part of the loop kernel. However, this situation changes after applying the transformation known as *skewing*: with this transformation, the whole domain is transformed, and the loop kernel, although it is equivalent to the original one, seems completely different. Now, in this new loop kernel, the iterations of the inner loop are independent, that is, can be executed in parallel. Thus, just by adding, for instance, an OpenMP dedicated pragma, the code can be parallelized.

Figure 4-3: Code and representation of the associated iteration domain: original loop kernel (left) and transformed loop kernel (right).

Thread-Level Speculation (TLS)

TLS [Steffan'98] –or *Speculative Parallelization*– is a well-known technique in code optimization and automatic parallelization consisting of optimistically executing regions of code –e.g., iterations of a loop– in parallel, assuming no dependences exist among them. While these regions are being executed, an underlying mechanism monitors every memory access to check no data races occur and, if so, halt all the involved threads and initiate a rollback to get back to a previous, non-faulty state.

TLS systems are usually composed of two main phases: a compile time and a runtime phase. During the former, some relevant properties of the code are extracted, such as data dependences that are already known, and some code snippets, dedicated to profiling and speculative execution, are statically generated. During the latter, the target code is instrumented on some execution samples to capture dynamic information. Once the parallel execution starts, the speculation is verified in order to ensure no conflicting accesses to memory addresses occur. However, if so, the faulty computations must be canceled and the memory restored to a previous, correct state, so this information must be stored somehow –e.g., with a backup of the memory region that is going to be affected–. Finally, in case of mispeculation, the faulty threads

are re-executed in a sequential fashion to overcome the conflicting memory access. Figure 4-4 provides an example –extracted from [Caamaño’16]– to illustrate how TLS works, where a simple kernel and two different execution traces are depicted.

Figure 4-4: Example of a TLS system: a) code and b) sequential execution (top) and speculative execution (bottom). Extracted from [Caamaño’16].

The first trace corresponds with a sequential execution, while the second represents the parallel execution of a TLS system. As can be noted, for the second trace, a first set of iterations is scattered among the four available threads. In this case, they access disjoint memory regions, so there is no dependence violation. Once this set is finished and the memory accesses verified, the changes are written to array A. After that, the process starts again, scattering a second set of iterations among the threads. However, now threads 3 and 4 access the same array element –A[9]–: first, thread 4 rewrites the value of this position, and afterwards thread 3 loads this value. This creates a so-called Write-After-Read (WAR) dependence violation, which is a data hazard that may occur when instructions exhibiting data dependences access/modify the same resource in different stages of a pipeline [Shen’13]. As a result, the first two threads are allowed to write their partial results, but the other ones are halted –rollback– and re-executed, this time sequentially, to overcome the dependence violation.

4.1.2 Apollo

As Apollo is based on TLS, it is also divided into two main parts, as shown in Figure 4-5: a static component, implemented as a set of passes of the Clang-LLVM compiler, and its dynamic counterpart, implemented as a runtime system. As can be observed, to use Apollo the designer only needs to: (i) enclose the targeted loop kernel(s) with a dedicated `#pragma` directive (`#pragma apollo dcop`); (ii) compile the code with Apollo’s own compiler, which is an alias over Clang built as a set of custom LLVM passes; and (iii) launch the resulting executable, which in turn triggers the runtime system of Apollo to orchestrate the execution and dynamically optimize the code.

Figure 4-5: Overview of Apollo. Extracted from [Caamaño’16].

The role of the static component of Apollo is to embed in the executable all the information that must be later available for the runtime system to correctly perform speculative parallelization. To that end, this component performs the following tasks:

- Optimize the code with LLVM classical optimizations. These optimizations are applied statically, i.e., to those parts that do not need runtime information.
- Accurately evaluate memory instructions that can be fully interpreted at compile time and embed this static information in the executable: type and number of memory instructions, structure of the loop kernel, aliasing information...
- Use a copy of the original code to build an instrumented version of the code to track those memory accesses that cannot be fully interpreted at compile time.
- Generate basic pieces of code, hereafter known as *code-bones* [Caamaño’17], which are units of code derived from the original code and expressed in their LLVM bytecode representation. Afterwards, at runtime, they are loaded and assembled to generate the final optimized code.

Code-bones are the solution Apollo proposes for fast runtime code generation, which is one of the main challenges of applying polyhedral optimizations at runtime. Code-bones are building blocks that, when instantiated and assembled, make possible to instantiate any polyhedral transformation at runtime. To do so, multiple code-bones are generated from the original code at compile time; then, at runtime, the desired transformation is built by instantiating and assembling these code-bones.

In speculative parallelization, two types of statements exist: the ones related to the computation itself, and the ones related to the verification of the speculation. Likewise, two types of code-bones exist: computation bones and verification bones.

- **Computation code-bones:** For each memory write instruction in the original kernel, a code-bone is derived. Each code-bone includes all instructions within the *backward static slice*, which consists of all the instructions required to execute the memory write, including memory read instructions. For example, the instruction $A[i][j] = B[i][j] + c$ performs one memory write, but its backward static slice is conformed by $B[i][j]$, i , j , c and all the previous instructions modifying any of these values. In other words, the backward static slice is a piece of code containing all the necessary information for executing a memory write.

- **Verification code-bones:** For each memory instruction of the computation bones –considering both read and write instructions– an associated verification code-bone is derived. Verification bones are also created for loop bounds and for the scalars –one for their initial value and another for the increment–. These code-bones are devoted to ensuring the validity of the speculation.

Once all the static information and all the code-bones have been embedded into the binary file, when this file is eventually executed, the runtime system of Apollo is launched and orchestrates the execution of the program.

To be reactive to possible changes leading to mispredictions, Apollo’s dynamic phase executes the loop kernel in *chunks* or slices. A *chunk* is a small set of complete contiguous iterations of the outermost loop. The loop kernel is thus executed in a succession of contiguous chunks; between them, the control of the workflow returns to the runtime system, which takes decisions about the execution of the next chunk, adapting itself to the different behaviors the applications being optimized may present. Figure 4-6 depicts the different decisions the runtime system can take, depending on the result of the execution of each chunk.

Figure 4-6: Transitions between behaviors of Apollo. Based on a figure from [Caamaño’16].

1. At the beginning of the execution of a loop kernel, or following the completion of a chunk running the original, sequential code, a **profiling** chunk is launched to observe and capture the behavior of the code at runtime.
2. Upon completion of the profiling, the **code generation** phase is started. If Apollo succeeds in generating the optimized code, this code is executed; if not, the remaining of the chunk is executed using the original version of the code, the prediction model is discarded, and the process starts again. Additionally, in parallel to the code generation, a background thread is launched with the original code, so as to partially mask the time overhead. An overview of the different tasks needed to finally generate the optimized code is provided in Figure 4-7. Specifically, during this phase, Apollo:

- (a) Builds a prediction model for the memory accesses and the loop bounds with the information obtained from the profiling –or instrumentation.
- (b) Loads, instantiates and assembles the code-bones to rebuild the kernel.
- (c) Feeds the linear model to Pluto compiler –previously described in Subsection 2.2.4 and used here as a library– so as to detect the dependences and generate an optimizing transformation.
- (d) Generates an optimized and parallel version of the original sequential code using the LLVM JIT compiler. As long as the prediction model still holds, the code generated by Apollo is semantically correct.

Figure 4-7: Overview of the code generation flow of Apollo. Extracted from [Caamaño'16].

3. Once the optimized code has been generated, and before actually executing it, a preventive memory **backup** is carried out to save the prior state of those memory locations predicted to be modified during the execution of the next chunk.
4. After the backup, chunks running the **optimized** code are executed until a misprediction –or mispeculation– is detected, or until the original loop exit is reached. It should be noted that, within optimized chunks, iterations are usually run in parallel threads. In order to continuously verify this prediction model, and thus the correctness of the program, Apollo implements a decentralized runtime verification system embedded in the parallel code. Each thread verifies independently if the address of the next memory area touched corresponds with the address predicted by the model. Such a verification strategy has considerably less overhead than centralized systems traditionally used in TLS [Steffan'05, Liu'06, Oancea'09, Raman'10].
5. If a misprediction happens, a **rollback** is launched to automatically retrieve the memory state prior to the execution of this chunk and saved in the backup.
6. After restoring the memory state, the same chunk is launched again with the **original** version of the code. Upon completion of this stage, if the original loop exit has not been reached yet, the previous prediction model is invalidated and the whole process starts again.

Finally, to illustrate how the execution by chunks of iterations works, Figure 4-8 presents an example of how the system evolves with respect to the elapsed time:

- At the beginning of the execution of a loop kernel, Apollo initially observes the original code for a small chunk –in this case, from iterations 0 to 7–, by executing the instrumented version of the code generated at compile time.
- Then, in the next chunk –iterations 8 to 37– the code generation phase is launched, together with the background thread running the sequential code that tries to mask the overhead this phase can add. This thread starts its execution from the last iteration the profiling phase completed.
- Once the optimized code has been generated, the thread executing the original code is halted, and the optimized code is launched instead, continuing the execution from where the sequential thread stopped –iterations 38 to 137.
- If a misprediction is detected, a rollback is initiated and the original code is executed again for the same chunk, since it is obviously semantically correct.
- After the completion of the faulty chunk, the prediction model is invalidated and the whole process starts again.

Figure 4-8: Succession of chunks, alternating among the different phases of Apollo. Extracted from [Caamaño’16].

4.2 Dataflow domain: PREESM and SPiDER

As stated in Chapter 3, PREESM is an open-source rapid prototyping design framework based on the Y-chart design [Pelcat’14]. It simulates applications and automatically generates code for heterogeneous multi/many-core systems. As shown in Figure 4-9, this framework requires three different input files: (i) a high-level description of the algorithm, (ii) a high-level description of the target architecture, and (iii) a scenario file gathering some user constraints, which are fed to its code generation mechanism. Although PREESM gathers a static backend, it also has a dynamic counterpart, known as SPiDER. Consequently, Subsection 4.2.1 will detail PREESM’s static behavior, while Subsection 4.2.2 will do the same with SPiDER.

4.2.1 PREESM

Figure 4-9: Overview of PREESM framework.

The high level description of the algorithm required by PREESM consists of a dataflow graph of the application implementing the PiSDF MoC, which was previously described in Section 2.1. As a reminder, this MoC is an extension of the SDF MoC, which models the application as a directed graph of computational units –*actors*– that exchange data packages –*tokens*– through a set of FIFO queues. PiSDF extends this model by adding two key features to the graph definition: hierarchy and parameters. On the one hand, the former increases the modularity of the MoC: with this addition, actors can now be composed, in turn, of a whole new graph comprising more actors. On the other hand, the latter allows users to either configure or modify the dataflow specification of an application easily, without having to modify the inner code of the actors. To illustrate these concepts, Figure 4-10 provides an example of an application graph, which in this case represents a *Sobel* filter, an image transformation widely used in image processing applications to detect edges of two dimensional images [Sobel’68].

Figure 4-10: Example of a PREESM application graph representing a *Sobel* filter. Extracted from PREESM website¹.

¹<https://preesm.github.io/about>

As can be seen, there are three actors, *Read_YUV*, *Sobel* and *display* –represented as gray boxes–, communicating through FIFOs – represented in turn as directed arrows connecting a production data port to its corresponding consumption data port–. The application presents also three parameters, *width*, *index* and *height*, represented as blue triangles. *height* and *width* are used to define the resolution of the images to which the *Sobel* filter is going to be applied, while *index* references the display window, so it can be used to enable simultaneous displays. These parameters are connected to the actors they affect through configuration ports. Additionally, each actor of the graph is associated with a C function or C++ method instantiated within the actor code.

Secondly, the high level description of the target architecture required by PREESM is based on S-LAM. This model represents parallel architectures as a set of PEs exchanging information through communication nodes and data links [Pelcat’09]. The main advantages of this model are that it (i) provides the developer with a basic description of the target architecture at system-level and (ii) reduces the complexity of the multicore simulation. Figure 4-11 provides an example of a 4-core architecture –*Core0*, *Core1*, *Core2*, *Core3*– in which the nodes communicate through a shared memory. This model is used during the code generation phase to map and schedule the actors on the available PEs and to route the inter-core communications.

Figure 4-11: Example of a PREESM S-LAM architecture graph denoting a platform with 4 cores.

Last, but not least, the scenario file gathers some user-defined constraints that delimit the final implementation and deployment of the system. This file is organized in different tabs related to configuring the application, as shown in Figure 4-12. Although there are many of them, the most relevant ones are *Constraints*, *Timings* and *Papify*, since the rest are devoted to configuring data types, output paths, etc.

- **Constraints:** As shown in Figure 4-12, in this tab, users can map each actor of the dataflow specification to the PEs in which the actor should be executed. This allows the user to limit the actors each PE can execute.
- **Timings:** Here, users predefine the amount of time each PE requires to execute each actor, which helps PREESM in refining the simulation of the different deployments and, thus, refining the mapping and scheduling of the application.
- **PAPIFY:** In this tab, the user can configure PAPIFY [Madronal’19], a monitoring toolbox that simplifies the profiling of dataflow applications by automatically retrieving hardware information from Performance Monitoring Counters (PMCs).

Figure 4-12: PREESM scenario. *Constraints* tab example.

Once the user provides PREESM with these three files, the code generation phase generates the specific code for the target architecture. The process PREESM follows to generate the code is gathered in the *workflow* file, an executable graph connecting rapid prototyping tasks, such as mapping and scheduling. All these tasks are implemented as Eclipse plug-ins, conferring high scalability upon PREESM. An example of a PREESM workflow is depicted in Figure 4-13. As can be observed, before getting to the actual code generation task, several steps are performed before. Among all of them, three should be highlighted: *Scheduling*, *Memory Allocation* and the actual *Code Generation* itself –marked in the figure by dashed red lines.

Figure 4-13: PREESM workflow.

- **Scheduling:** It deals with the mapping and scheduling of the application onto the available PEs of the target architecture. Up until now, PREESM only includes latency-based algorithms, whose objective is to devise implementations that optimize the latency of the application. If several actors can be executed simultaneously and there are enough PEs available, they can be mapped to different PEs, thus parallelizing their execution.

- **Memory Allocation:** This task deals with withholding on each PE the necessary memory to guarantee the system execution –i.e., for allocating the input/output data required to complete the execution–. It is crucial for ensuring a bounded memory execution, one of the key features of a decidable MoC. If this cannot be guaranteed, unexpected deadlocks may appear if a FIFO runs out of memory. Thus, PREESM generates deadlock-free code for PiSDF applications.
- **Code Generation:** This task uses the results of the previous steps to generate compilable code for the target architecture. The code generation step produces a static code for each available PE in the target architecture with automatic cache management, synchronization and inter-core communication [Sriram’18].

Before concluding this section, a last, yet very important characteristic of PREESM should be described. As mentioned at the beginning of this section, PREESM is able to automatically generate code for heterogeneous multi/many-core systems. This means that, apart from generating code for each of the available PEs in the target architecture, PREESM should have some mechanism to expose actor parallelism, so as to be able to exploit the parallelism possibilities offered by a multi/many-core system.

To illustrate how to do this through PREESM, the application shown in Figure 4-10 is going to be used. However, this application is sequential, and it needs to be slightly modified so as to explicitly expose a parameterizable degree of data parallelism. As described in [Georgakarakos’17], *Sobel* filter can be parallelized by splitting the original image to be filtered into several slices that can be processed in parallel and recombined afterwards. To do so, the network shown in Figure 4-10 is modified as depicted in Figure 4-14. As can be observed, two new actors have appeared, *Split* and *Merge*, to deal with splitting the image into slices and re-merging the slices into a whole image after their processing. Likewise, two new parameters have appeared, *nbSlice* and *sobel_height*, to set the number of slices in which the image is going to be divided and to define the height of each slice, respectively.

Figure 4-14: Example of a PREESM application graph representing a *Sobel* filter with *Split* and *Merge* actors to split the processing of the application. Extracted from PREESM website².

²<https://preesm.github.io/tutos/parasobel>

Now, with this kind of application graph, the parallelism can be exposed thanks to the production and consumption rates of the data ports of the actors involved in the computation. For a PREESM application to work properly, the production and consumption rates of the actors' data ports must match, i.e., that the tokens generated by one actor are consumed by another. This means that, if an actor generates more tokens than the actor to which it is connected consumes, PREESM will try to map as many instances of that actor as needed to match this production rate. This behavior is shown in Figure 4-15: if actor *Split* from Figure 4-14 produces eight tokens and actor *Sobel* from the same figure consumes only one, the behavior of PREESM will be the one depicted in Figure 4-15, where eight instances of actor *Sobel* are created. It should be noted that, for this mechanism to work properly, actor *Merge* needs to consume eight tokens, accordingly to the production and consumption rates of the previous actors. Likewise, two automatic actors appear in the figure, *explode_Split_0_output* and *implode_Merge_0_input*, which deal with sharing and retrieving, respectively, the data tokens among the eight instances of actor *Sobel*. Then, each instance of the actor can be mapped to a different resource, hence providing another level of parallelization to the actor, which can, in turn, be adjusted by the parameters of the network.

Figure 4-15: Example of a PREESM network graphically exposing the parallelization of *Sobel* actor. Extracted from PREESM website³.

³<https://preesm.github.io/tutos/parasobel>

4.2.2 SPiDER

As described at the beginning of this chapter, SPiDER is a dataflow-based runtime resource manager that targets multicore heterogeneous platforms. The objective of SPiDER is to allow efficient and dynamic reconfiguration of applications by taking advantage of the PiSDF MoC properties. This runtime manager was born to support reconfigurability within PREESM framework, which comes by means of detecting and supporting dynamic changes in the parameters of the application.

Considering the *Sobel* application presented in Figure 4-14, Figure 4-16 shows the evolution of the static dataflow specification to become a dynamic one. In this application, as can be observed, there is a new actor, together with two new parameters, which all have in common a small white circle on their upper side. This symbol indicates, in the case of parameters, that they are dynamic; in the case of the actor, that it is not a computation actor, but a configuration one. The main difference between them is that the latter is only devoted to updating the value of dynamic parameters and, thus, presents special firing rules. Configuration actors are only fired once, at the beginning of the execution of the graph including them. With respect to the dynamic parameters, the two of them are devoted to configure the number of slices in which the input image is going to be split. Hence, by reconfiguring these parameters at runtime, the parallelism of the application is also dynamically reconfigured.

Figure 4-16: Example of a SPiDER application graph representing a *Sobel* filter. Extracted from PREESM website⁴.

Consequently, SPiDER takes the same three input files as PREESM: (i) a PiSDF application graph, (ii) a high level description of the target architecture following the S-LAM model, and (iii) the scenario file with the user-defined constraints. These files are fed to SPiDER workflow, which is shown in Figure 4-17. As can be seen, this workflow seems apparently simpler, since everything is performed at runtime.

⁴<https://preesm.github.io/tutos/spider>

Figure 4-17: SPiDER workflow.

To take scheduling decisions at runtime, SPiDER is divided into two separate engines: a GRT and several LRTs. The GRT analyzes the application topology, takes scheduling decisions and distributes the work over the available PEs of the target architecture, considering to that end the values of the dynamic parameters, which may affect the scheduling or the work distribution. In contrast, the LRT are lightweight runtimes devoted to execute the actors sent to them.

Figure 4-18 exemplifies SPiDER functioning when two LRTs are used. First, the GRT analyzes the application topology and decides the scheduling of the actors conforming the graph over the available PEs ❶. After, the GRT sends to each LRT their corresponding jobs ❷. The LRTs, in turn, execute their jobs –i.e., their associated actors– ❸ once the required tokens become available in the pool of data queues –or FIFOs– ❹. When the execution of each actor finishes, each LRT sends some information back to the GRT, namely the new value of the dynamic parameters ❺ and some trace information for debugging purposes ❻. With this information, the GRT revisits the scheduling, modifying it if necessary for the next iteration of the graph.

Figure 4-18: SPiDER runtime structure. Based on a figure from [Heulot'14].

Once the tools involved in this PhD have been introduced, next chapters will present the contributions to each of them, both separately and as a whole.

5

DATAFLOW-BASED RUNTIME POLYHEDRAL OPTIMIZATIONS

As mentioned in Chapter 1, dataflow MoCs are a good solution for exploiting the structural parallelism of applications while maintaining a high level of abstraction, which in turn helps developers in increasing their productivity. However, there are some drawbacks, namely a poor workload distribution or the presence of compute-intensive actors, which can lead to a drop in the expected performance and, therefore, to not fulfilling the performance requirements of the application.

In that regard, the solution proposed in this PhD work to overcome this drawback is to extend the Y-chart, dataflow-based design methodology as depicted in Figure 5-1. This proposal aims at taking advantage of the iterative nature of stream-based, data-driven applications to evaluate different possible optimizing transformations and select the best performing one among them. To do so, the approach proposed in this work is to embed the polyhedral domain into the Y-chart, dataflow-based design methodology, as shown in Figure 5-1.

Additionally, nowadays many applications need to be reactive to changes at runtime, so many characteristics cannot be known in compile-time. As a result, most polyhedral tools are not very useful in these contexts, since, as seen in Section 2.2, the great majority of the state-of-the-art polyhedral tools only work at compile time. For that reason, Chapter 4 established Apollo as the starting point of this work from the polyhedral domain point of view, as it is one of the few tools applying the polyhedral model at runtime and it already implements several features that can come in handy for combining a polyhedral tool with a dataflow framework.

Although very powerful, polyhedral tools cannot always maximize the performance of applications. As they are usually conceived as general-purpose tools or mainstream compilers, they normally apply those transformations already proven to provide the best outcomes in general terms. However, depending both on the application and the target architecture, the performance achieved by this generic solutions might not be the maximum achievable. For instance, if two architectures have different cache memory sizes, it cannot be expected for the optimizing transformation to yield the same performance in both of them; likewise, if two applications present different memory usages –e.g., very small or very large input data sizes–, it seems unlikely that the same transformation results in the best performing one for both cases.

However, when the usage of these tools is limited to just one type of applications, such as those dataflow-based, the techniques applied to improve the performance can be also narrowed down to take advantage of the nature of these applications to specialize the optimizations applied and maximize the performance achieved.

Figure 5-1: Overview of the improved design methodology proposed in this document to combine the Y-chart design methodology with polyhedral optimizations.

Therefore, one of the goals of this work is to propose an optimizing polyhedral framework as the one depicted in Figure 5-2, which provides a detailed view of the part of the design methodology embedded into the polyhedral domain –i.e., the yellow box in Figure 5-1–. The objectives of this optimizing polyhedral framework are: ❶ to generate on-the-fly several transformed versions of loop kernels that are relaunched many times; ❷ to evaluate and compare these versions in terms of performance; ❸ to store the best version; and ❹ to automatically retrieve and launch this version whenever the same loop kernel is revisited. Additionally, as already said, applications may change at runtime depending on certain external parameters, so this framework needs to be aware of the state of those parameters, and ❺ reconfigure itself accordingly.

Figure 5-2: Detail of the extension of the Y-chart design methodology through a runtime multiversioning system based on polyhedral transformations.

Hence, this chapter details the main contributions of this thesis to a polyhedral tool like Apollo to explain how this framework has been extended to include the new functionalities. To do so, the organization of this chapter is the following: first, several versions must be generated and tested –Section 5.1–; then, a mechanism to compare these versions in terms of performance should be defined –Section 5.2–; once the versions are compared, the best version has to be stored –Section 5.3–. Additionally, a mechanism to generate new versions if the same version does not hold for the entire execution of the loop kernel needs to be devised –Section 5.4–. Furthermore, the best version of each kernel needs to be associated to the state of the external parameters, so as to start again if the system is reconfigured –Section 5.5–. Finally, a summary of all the contributions is also provided to depict how the whole framework works –Section 5.6.

5.1 Multiversioning

As described in Chapter 3, one of the main ideas of the work presented in this document is to build a runtime multiversioning system that takes advantage of the iterative nature of dataflow applications to generate several optimized versions of compute-intensive kernels, evaluate them and apply the best one for each case. The rationale behind this is to be able to test different polyhedral transformations, as each one may be more or less beneficial, depending on the circumstances.

In the context of code optimization and compilation, multiversioning is a well-known technique to generate code that may adapt to a changing execution context: first, several versions of an original code snippet are generated, each one testing

different optimizing transformations. Afterwards, when the resulting code is launched, certain runtime decisions are taken so as to select the most suitable version to be run.

Nonetheless, static multiversioning—all the versions are generated at compile time—is only efficient when all future execution contexts are known beforehand, but when they are mostly unknown or unpredictable, this kind of approach becomes inefficient. Runtime multiversioning—generating versions at runtime—could be the answer to such limitations. However, such a runtime approach usually comprises several time-consuming steps: profiling, analysis, transformation, and just-in-time compilation.

One of the reasons why Apollo has been chosen as the reference polyhedral tool is because it is a dynamic, optimizing framework already containing all these steps. Although this tool has already been described in Section 4.1, a brief summary of how it works and how it implements those steps is also provided here. As stated before, Apollo is a polyhedral framework for applying parallelizing and optimizing transformations on-the-fly to loops exhibiting a polyhedral-compliant behavior at runtime [Caamaño’17]. It is based on TLS [Steffan’98], an approach to overcome one of the main limitations of automatic parallelization techniques, which is the limited information available at compile time. This approach consists of executing optimistically—or *speculatively*—regions of code in parallel, assuming that no dependences ever exist among them. However, for this technique to succeed, there must be an underlying mechanism to monitor the execution so as to ensure no data races occur and, if so, initiate a rollback, which recovers the state prior to the memory violation. Considering all this, Apollo is divided into two main parts:

- **Static compiler:** This compiler, implemented as a set of passes of the Clang-LLVM compiler¹, analyzes each target loop nest in terms of memory accesses, loop bounds and the evolution of its scalar variables, and classifies them as static or dynamic. This classification is then used by the compiler to build an instrumented version of the code, in which some instructions are inserted so as to collect both the actual values of dynamic objects and the initial values of static objects when executed—i.e., base addresses of regular data structures.
- **Runtime system:** The runtime manager of Apollo orchestrates the execution of the program. To do so, the execution of each loop kernel follows a set of successive phases, each one corresponding to the execution of a slice—or chunk—of iterations of the outermost loop:
 1. Instrument the code: In this phase, a small set of iterations is executed sequentially to record memory addresses, parameter values, etc.
 2. Compute an optimizing transformation: The information retrieved in the previous phase is used to build a prediction model for the loop kernel, which is fed to Pluto, here used as a library, to generate the optimizations to apply. After Pluto determines the transformation, the corresponding

¹<http://llvm.org/>

parallel code is generated by the LLVM JIT compiler, including some additional instructions to check if the prediction model is still valid.

3. Run the optimized code: During this phase, a large slice of iterations is executed applying the optimized transformation determined by Pluto. However, before this execution starts, a backup of the memory regions predicted to be updated in that execution is performed, so as to retrieve the previous state if a rollback occurs. While executing the optimized code, the prediction model is constantly verified by checking that the actual reached values and their predictions match.
4. Handle mispredictions: When the execution of the slice finishes, if no misprediction was detected, a new backup of the next slice is stored, and the next slice is executed with the same optimized code. On the contrary, if a misprediction occurs, the previous memory state is restored through the backup and the execution of the slice is restarted, this time with the original version of the code, so as to overcome the execution point causing the misprediction. Afterwards, a new profiling slice is launched to build a new prediction model that captures the changing behavior.

However, although Apollo has been proven to be a powerful tool for applying polyhedral optimizations at runtime, one of its main drawbacks is that it is blind to the number of times the same kernel is executed. As shown in Figure 5-3, which summarizes Apollo's dynamic phase, if the same kernel is ever executed again, the same steps are always repeated, re-instrumenting the code and, even worse, re-generating always the same transformation.

Figure 5-3: Apollo diagram.

Even though Apollo presents mechanisms for the user to guide Pluto in the transformation selection –i.e., options to activate/deactivate polyhedral transformations, such as tiling or unrolling–, these guidelines are common for the entire execution of a program. That means that all the loop kernels Apollo could handle within a program would be treated equally, applying those same guidelines to all of them, regardless of

their specific features, shapes, etc. This means that, regardless of the number of kernels processed by Apollo, and regardless of the number of times the same kernel has to be launched, the same optimizing transformation is always generated and tested.

This missed opportunity becomes especially relevant when considering the nature of dataflow systems, which implement applications performing the same processing to a stream of data, thus running on a loop and invoking the same kernels again and again, iteratively. Additionally, systems working with data streams as inputs are usually required to yield, at least, an apparent real-time behavior. Hence, this drawback has an even greater impact, since it can cause the system to not reach the desired throughput.

For those reasons, an extension of Apollo in order to take advantage of these new features is proposed: to build a runtime multiversioning system whose goal is to generate and evaluate on-the-fly several transformed versions of loop kernels that are launched iteratively, as is the case of stream-based, data-driven applications. Consequently, at each invocation of a given kernel, the new framework either generates, launches and evaluates a new, transformed version, or directly launches the one that has been previously selected as the best performing version for that kernel.

As a result, the new runtime multiversioning system, depicted in Figure 5-4, is organized in two distinct phases: the training phase and the operational phase.

Figure 5-4: Overview of the different phases of the dynamic multiversioning system: training phase (left) and operational phase (right).

For each loop kernel processed by Apollo, first the training phase –Figure 5-4a– is successively relaunched until every polyhedral transformation, among a fixed set of transformations, has been generated and tested. Then, the best version is elected by

comparing all the tested versions in terms of performance –see Section 5.2–. As kernels may be launched iteratively, in the operational phase –Figure 5-4b– the versions elected as best performing will be the ones being generated and launched directly, thus applying directly the optimizing transformation selected as the best one.

To implement these new features within Apollo, its dynamic phase has been modified and extended as depicted in Figure 5-5, where the new parts appear in purple, while the original ones are denoted in green. If it is the first time the loop kernel is ever launched, Apollo behaves as described in Section 4.1, generating an optimizing transformation and executing the optimized code for a chunk of the outermost loop. The main difference here with respect to the behavior described in Section 4.1 is that Apollo has been modified to apply one optimizing transformation among a fixed set of transformations, instead of using Apollo’s default configuration. Each transformation is defined by a specific set of flags used to invoke Pluto compiler, which are related to activating or deactivating some polyhedral transformations, such as tiling or unrolling.

Figure 5-5: Diagram of Apollo’s dynamic phase with multiversioning.

The group of flags that can be used to configure Pluto is shown in Table 5-1. By activating one or more, one guides Pluto in generating an optimizing transformation, which in turn can comprise one or the combination of several polyhedral transformations –e.g., tiling + interchange–. Hence, different transformations can be generated varying the combination of these flags, being more or less beneficial depending on factors related to the loop kernel –data dependences, memory footprint...– but also with respect to the target hardware platform. This can have a great impact on the performance of the tested transformation, since the most efficient transformation for a specific set of conditions can be much less efficient if those conditions vary. One of the most representative examples is the tile size: for different cache memory hierarchies or input data sizes, the tile size providing the best performing code may vary.

Table 5-1: List of flags to configure the polyhedral transformations applied by Pluto.

Flag	Description
<code>--tile</code>	Activate tiling
<code>--intratileopt</code>	Optimize intra-tile execution, allowing transformations as loop interchange
<code>--l2tile</code>	Apply tiling a second time (usually for optimizing L2 cache usage)
<code>--diamond-tile</code>	Apply load-balanced tiling, specifically thought for stencil computations
<code>--parallel</code>	Parallelize automatically (insert OpenMP directives)
<code>--rar</code>	Consider Read-After-Read (RAR) dependences, not only WAR and Read-After-Write (RAW)
<code>--unroll</code>	Activate loop unrolling
<code>--ufactor</code>	Factor controlling loop unrolling
<code>--prevector</code>	Enable transformations to make code amenable to vectorization
<code>--identity</code>	Parallelize the loop kernel by slicing each outermost loop into parallel threads, without applying any further transformation

As a result, the new runtime multiversioning system generates and tests different transformations from a fixed set of flag combinations. Although the number of flag combinations is initialized to a default value within Apollo, it can be easily modified and customized by the user, as will be further addressed in Chapter 6. Likewise, the specific combinations selected for the experiments carried out in this document will be described in Section 8.1.

During the training phase, to evaluate the different versions, some information related to the tested optimizing transformation must be stored. This information is presented in Table 5-2; as can be noted, at least two identifiers are required, one per kernel –*Loop ID*, in case several kernels are handled by Apollo– and one per optimizing transformation –*Transformation ID*, to keep track of the already evaluated transformations–. The former is directly provided by Apollo’s static phase, which already creates a unique identifier of the pragma associated to the loop kernel triggering Apollo’s runtime phase; on the contrary, the latter did not exist until now and, thus, has been created within Apollo’s runtime phase. Additionally, if the objective is to compare the versions in terms of performance, some kind of metric associated to the transformation must be also stored. This issue will be addressed in Section 5.2.

Table 5-2: Information that needs to be stored for each optimizing transformation.

Item	Description
Loop ID	Unique identifier of the handled kernel
Transformation ID	Unique identifier of the transformation being tested
Execution time	Measured execution time of the transformation being tested

If the same kernel is ever relaunched, Pluto is called again, but this time with a different flag combination, among the fixed set of combinations, to generate and test a new transformation. When the execution of this iteration is finished again, the last tested transformation is compared with the stored one regarding its execution time. If the new one is more efficient in terms of performance, it substitutes the saved one. Otherwise, the only information stored with respect to the already discarded transformation is its identifier to allow keeping track of the transformations already tried. This process is repeated each time the same kernel is invoked and until all flag combinations have been tested. When this occurs, the training phase ends, and the operational phase begins. During this phase, the flag combination selected as the best one is directly fed to Pluto, avoiding testing more transformations.

Although this scheme already provides a new degree of optimization with respect to the starting point, there are still some missed opportunities, and some issues that should be addressed, as how to find a suitable metric for assessing the performance of each version to properly compare them –Section 5.2– and how to take advantage of the technique known as memoization –Section 5.3.

5.2 Evaluation of the best versions

In the previous section, the basic scheme for building a dynamic multiversioning system was outlined. However, in order to build a robust system, the methodology to compare two versions must be correctly defined. As already mentioned, since the objective of such a system is to find the transformation providing the best performance, minimizing the execution time will be the main criterion. To compare transformations in terms of performance, one could think that just measuring the execution time of each version and keeping the fastest one could be a sufficient metric to assess the performance of the version under evaluation.

Yet, as described in Section 4.1, the optimizing transformation is not launched for the entire execution of the kernel, but just for only a slice or chunk of iterations of the outermost loop. Although the size of this chunk is always the same –either set at the beginning of the execution by the user or initialized to a default value–, there are some cases in which this size is reduced: for instance, when a misprediction is detected, before discarding the prediction model, first the optimization is retried for a smaller chunk, just in case the optimization is still valid for a few iterations. In that case, if

the comparison of versions is performed just in terms of their global execution time, the result would not be reliable, since one would be comparing two versions that have been executed for a different number of iterations, and thus, have a different workload.

A possible workaround to solve this issue could be using as a metric the time per iteration of the outermost loop. Since the optimized version is executed in chunks, the beginning and ending iterations are known at runtime, so the metric could be to measure the execution time of the chunk and divide it by the total number of iterations of the outermost loop –i.e., subtracting both values–. Nonetheless, this approach presents two main issues that prevent it from being the most suitable one.

The first issue is related to the mechanism Apollo includes to counterbalance the overhead of performing some computations at runtime –e.g., to mask the overhead induced by the usage of Pluto at runtime–. To do so, Apollo launches a sequential thread running some iterations of the original kernel while another thread deals with the profiling and transformation selection phases. When these phases finish and Pluto generates the transformation to apply, the sequential thread is stopped. Then, the chunk executing the optimized code is launched, but its starting point is set to the last iteration the sequential thread completed. As a result, it is not possible to predict the iteration in which the optimized chunk is going to start its execution, since it is directly linked to the time invested by Pluto in generating the optimizing transformation. Although this issue does not prevent the previous metric to be successful –in any case, the total number of iterations is known–, it is directly linked to the second issue.

The second issue is related to the shape of the loop kernel, which depends on the iteration domain. This concept is already outlined in Section 2.2, but a brief summary is presented here. The iteration domain of a statement within a loop nest represents the values that the iterators of the loops surrounding each statement can take. To illustrate this concept, Figure 5-6 presents two different loop nests, which will be used to show the different shapes a domain can take; additionally, it also presents the graphical representation of the iteration domains associated to each of the loop nests. As can be observed, the iteration domain of the first loop nest is rectangular, while the domain of the second one is triangular. Non-rectangular iteration domains appear when some loop bounds depend on the surrounding loop iterators, as happens in the second loop nest with loop iterator j . In such cases, the problem is that the number of iterations for a slice or chunk of the outermost loop is not constant.

This can be observed in Figure 5-6: for both domains, two different chunks of the outermost loop –in this case, i – have been represented. In both cases, chunk 1 goes from $i = 1$ to $i = 2$, while chunk 2 goes from $i = 3$ to $i = 4$. Considering that white circles represent iterations, one can easily notice that, for the rectangular domain, the number of iterations per chunk (10) is constant for any outermost loop bound. On the contrary, in the triangular domain, chunk 1 gathers 7 iterations, while chunk 2 includes only 3.

Hence, if the execution time divided by the number of iterations of the outermost loop were to be used as a performance metric, it would only be accurate for rectangular iteration domains. For any other iteration domain, that figure of merit would not correctly represent the performance of the version.

Figure 5-6: Example of different domains: a) rectangular and b) triangular. Two chunks are also represented as an example.

Consequently, a more generic figure of merit should be found. In that sense, the proposal is to compute the exact number of iterations that have been executed for any slice of the outermost loop, whichever the shape of the iteration domain and the loop bounds. To avoid the time overhead that would add the increase of a dedicated counter, a feasible solution is to compute this number directly through the evaluation of the associated Ehrhart polynomial [Claus's'96].

As defined in [Lazcano'20], Ehrhart polynomials express the exact number of integer points within a parameterized polyhedron. They can be computed using existing tools, as the `barvinok` library [Verdoolaege'07b]. Unfortunately, invoking such a library at runtime would introduce an overhead impossible to mask. As a result, the runtime computation of the Ehrhart polynomial associated to each loop kernel has been implemented within the dynamic multiversioning system proposed in this work.

Following the theory presented in [Claus's'96], for a loop nest of depth d –with d being the number of nested loops–, its Ehrhart polynomial is a polynomial whose variables are the lower and upper bounds of the outermost loop, and whose degree is at most d . The general form of the Ehrhart polynomial is presented in Equation 5-1, where $ep(lower, upper)$ represents the number of iterations when the outermost loop index goes from $lower$ to $upper$, d is the depth of the handled loop kernel, $lower$ and $upper$ are the possible values of the outermost loop index and c_{ij} 's are the unknown rational coefficients that have to be found to generate the Ehrhart polynomial.

$$ep(lower, upper) = \sum_{i=0}^d \sum_{j=0}^{d-i} c_{ij} lower^i upper^j \quad (5-1)$$

The number of unknown coefficients to compute is given by Equation 5-2, which is also the number of linear equations that have to be built to get a solvable system.

$$\#unknowns = \frac{d^2}{2} + \frac{3d}{2} + 1 \quad (5-2)$$

To build this system of $\#unknowns$ equations, $\#unknowns$ different valid combinations of $lower$ and $upper$ need to be used. For each of these combinations, the associated number of iterations run by the handled loop nest –i.e., $ep(lower, upper)$ – must be computed. This computation can be performed using the expression presented in Equation 5-3, where $c(x)$ represents the number of iterations executed by the handled loop nest when its outermost loop index is equal to x .

$$ep(lower, upper) = c(lower) + c(lower + 1) + \dots + c(upper) = \sum_{x=lower}^{upper} c(x) \quad (5-3)$$

Hence, to be able to compute this number, one must previously obtain the number of iterations executed within an iteration of the outermost loop –i.e., $c(x)$ –. This number could be obtained by either increasing a dedicated counter, which would add an overhead, or by computing it beforehand. To show how this can be done, a generic form of a loop nest of depth 3, provided in Figure 5-7, is used.

```

for(i = lower; i < upper; i++)
    for(j = 0; j < f(i); j++)
        for(k = 0; k < g(i, j); k++)
            // Computation
    
```

Figure 5-7: Generic form of a loop nest of depth 3.

In this generic form, the loop bounds have been represented as linear functions of their surrounding loop iterators: j linearly depends on i , and k linearly depends on i and j . This linear relationship is gathered in Equation 5-4 and Equation 5-5.

$$f(i) = a \cdot i + b \quad (5-4)$$

$$g(i, j) = c \cdot i + d \cdot j + e \quad (5-5)$$

As the objective is to calculate how many iterations correspond with an iteration of the outermost loop, the generic forms of $f(i)$ and $g(i, j)$ are going to be evaluated to a specific value of i , $i = lower$. The result of this evaluation is provided in Equation 5-6 and Equation 5-7. As observed, now $f(lower)$ is a constant, N , and $g(i, j)$ can be expressed as $d \cdot j + Y$, Y being another constant.

$$f(lower) = a \cdot lower + b = N \quad (5-6)$$

$$g(lower, j) = c \cdot lower + d \cdot j + e = d \cdot j + Y \quad (5-7)$$

Now, $c(lower)$ can be obtained by accumulating all the possible values $g(lower, j)$ can take, as shown in Equation 5-8.

$$c(lower) = \sum_{j=0}^{f(lower)} g(lower, j) = \sum_{j=0}^N (d \cdot j + Y) \quad (5-8)$$

This expression can be easily simplified using Equation 5-9, which represents the sum of the simplest arithmetic progression, composed by the N first natural numbers.

$$\sum_{j=1}^N j = N \cdot \frac{(N+1)}{2} \quad (5-9)$$

As a result of combining Equation 5-8 and Equation 5-9, Equation 5-10 is straightforwardly obtained.

$$c(lower) = Y \cdot (N+1) + d \cdot N \cdot \frac{(N+1)}{2} \quad (5-10)$$

Now $c(lower)$ can be directly computed, as all the variables involved can be directly extracted from the profiling phase—all of them are related to the loop bounds—. For example, if the loop nest being handled by Apollo is the one showed in Figure 5-8, now $f(i)$ and $g(i, j)$ should be expressed as in Equation 5-11 and Equation 5-12.

```
for(i = 0; i < 9; i++)
  for(j = i; j < 2; j++)
    for(k = j; k < 4; k++)
      // Computation
```

Figure 5-8: Example of a loop nest and its related iterators.

$$f(i) = -i + 2 \quad (5-11)$$

$$g(i, j) = -j + 4 \quad (5-12)$$

And, consequently, if one were to compute the number of iterations for $i = 0$, the evaluated expressions would be as those of Equation 5-13 and Equation 5-14, and, thus, the value of the variables appearing in Equation 5-10 would be those shown in Equation 5-15. Finally, evaluating Equation 5-10 with these numbers, one would obtain that the number of iterations for $i = 0$ is 9, which is easily provable by hand.

$$f(0) = 2 \quad (5-13)$$

$$g(0, j) = -j + 4 \quad (5-14)$$

$$N = 2, Y = 4, d = -1 \quad (5-15)$$

This process has been implemented within Apollo. Considering that the maximum loop depth Apollo can handle is set to 6, a mechanism has been designed to compute the generic forms of Equation 5-10 for loop depths up to 6, and to evaluate these forms with the values of the loop bounds obtained through the profiling phase, so as to build this system of *#unknowns* equations.

Once the system of linear equations has been built, it can be easily solved by invoking a solver already existing within Apollo, which is based on *Armadillo*, a high quality linear algebra library whose objective is to find a good balance between speed and ease of use [Sanderson'16]. As a result, from this system of linear equations, the Ehrhart polynomials associated to each loop nest handled by Apollo are built, and whose variables are the *lower* and *upper* values of the slice being computed. Then, once the specific slice of iterations of the outermost loop starts to be executed, the polynomial can be evaluated, as *lower* and *upper* will have a known value, finally obtaining the exact number of iterations that are going to be executed within that slice.

In conclusion, the runtime multiversioning system proposed in this document includes the computation and evaluation of the Ehrhart polynomial associated to each handled loop kernel to compute beforehand the total amount of iterations executed within a slice of iterations of the outermost loop. Then, by dividing the execution time measured for that slice by the related and evaluated Ehrhart polynomial, the average execution time per iteration is obtained, which is a figure of merit that can now be used to accurately and robustly compare two optimized versions.

5.3 Memoization

Up until now, a new runtime multiversioning system to take advantage of the nature of stream-based, data-driven applications has been presented. As presented in Figure 5-5, this system is divided into two different phases, the training phase and the operational phase. During the training phase, each time the same kernel is launched, different polyhedral transformations are generated and evaluated with the metric presented in Section 5.2, so as to select the best performing transformation for that loop kernel; when a fixed number of transformations has already been tested, the training phase ends, leading to the beginning of the operational phase. During this phase, whenever the same loop kernel is ever launched, the selected transformation is directly generated, without having to test any more transformations.

This functionality supposes a significant improvement with respect to how Apollo worked before, when no matter how many times the same loop kernel was launched,

the same optimizing transformation was always generated, leaving out significant optimization opportunities. However, this is not the only drawback arisen by trying always the same transformation: it also yields useless time-overheads, since the profiling and code generation steps could be executed just once for all invocations of the same loop kernel, instead of doing it every time. To do so, the runtime multiversioning system should keep this information throughout all the executions of the loop kernels, so as to directly reuse it whenever the same kernel is relaunched.

To do this, memoization could be a suitable solution. As described in [Lazcano'20], memoization [Michie'68, Hall'97, Acar'03] is a popular optimization approach to accelerate functions by saving the results of calls –and linking them to the specific inputs– and returning the cached result when the same inputs occur again. Considering the specific context of the runtime multiversioning system described in this chapter, memoization could be applied in such a way that all the information related to each loop kernel is stored in order to be directly retrieved whenever the associated execution context occurs again.

Including memoization into the scheme of the runtime multiversioning system results in several performance improvements. Until now, with the multiversioning scheme, some information needed to be stored, namely a unique identifier per loop kernel and transformation and the execution time per iteration –see Table 5-2–. With this new approach, the information stored per loop kernel is extended. As highlighted in yellow in Table 5-3, besides the information already needed by the multiversioning system, now the instrumentation results, the prediction model and the optimized code, generated by the LLVM JIT compiler, are also stored.

Table 5-3: Information that needs to be stored for each optimizing transformation, evolving from Table 5-2.

Item	Description
Loop ID	Unique identifier of the handled kernel
Transformation ID	Unique identifier of the transformation being tested
Execution time	Measured execution time per iteration of the transformation being tested
Profiling results	Results generated during the profiling phase
Prediction model	Prediction model built after the profiling phase
Optimized code	Optimized code generated by the LLVM JIT compiler

By saving these results and keeping them throughout the execution of the application, the performance improvement is twofold:

- **Profiling phase:** As described in Section 4.1, whenever a kernel is handled by Apollo, a few iterations of the outermost loop are executed sequentially in order to retrieve the information needed to build the prediction model. This computation is not negligible, so there is an overhead that is being added every time Apollo is called. In the context of applications launching the same loop kernels iteratively, this overhead has a larger impact on the performance, since

it is repeating the same computation each time a kernel is handled. As a result, if both the instrumentation results and the prediction model are stored and retrieved whenever the same kernel is launched again, their associated computation would only be performed once, instead of each time the kernel is executed, thus resulting in a performance improvement.

- **Code generation phase:** As also outlined in Section 4.1, once Pluto finishes its execution and generates the transformed code, its result is fed to the LLVM JIT compiler to regenerate the binary to include the runtime changes. Although Apollo implements a state-of-the-art mechanism to reduce the overhead of this operation by recompiling only small pieces of the code –*code-bones*–, its overhead cannot be considered negligible. Therefore, if the binary code associated to the best transformation is stored, it can be automatically retrieved once the training phase ends, providing directly the best performance, without any further code analysis or transformation. Consequently, the operational phase is considerably simplified with respect to what was shown in Figure 5-4, since now it simply consists in loading and launching the best version associated to the current and already encountered loop kernel.

Figure 5-9 gathers an overview of how the implementation of the system has evolved from Figure 5-5 by combining multiversioning and memoization. As can be seen, the call to the JIT compiler has now disappeared from the new functionalities –represented in purple– and the instrumentation phase is executed only once per kernel, which are the main improvements offered by memoization in this context.

5.4 Rollback mechanism

Up until now, the standard execution flow of the runtime multiversioning system has been outlined. However, as detailed in Subsection 4.1.2, Apollo implements a speculative approach that makes possible the application of polyhedral optimizations to loop kernels that cannot be handled statically, but which exhibit a polyhedral-compliant behavior at runtime. However, this comes at the cost of having to resort to speculative parallelization, which in turn requires an underlying mechanism to monitor the execution so as to ensure no data races happen and, if so, launch a rollback, which recovers the state prior to the memory violation.

Therefore, the runtime multiversioning system depicted in here should also be able to deal with loop kernels that cannot be handled statically, so the mechanism to handle rollbacks should also be supported.

A rollback may occur while testing one of the optimization candidates, i.e., when executing the transformed code inside a chunk or slice of the original outermost loop. For such cases, the mechanism to handle rollbacks within Apollo behaves as follows:

Figure 5-9: Diagram of Apollo's dynamic phase with multiversioning and memoization.

- If the optimized code running when the rollback occurs was successfully run in a previous chunk, the same optimized code is launched again for a smaller chunk.
- If a rollback occurs again in this smaller chunk, or if the optimized code was actually never successful, the original serial code is launched for the next chunk.
- Once the faulty chunk finishes its execution, the whole process starts again for the remaining iterations: instrumentation, transformation selection and optimized code execution. Instrumentation must be executed again because the rollback invalidates the prediction model, since there has been a misprediction.

To be reactive to possible rollbacks, the runtime multiversioning system has been modified to behave as shown in Figure 5-10. During the training phase (Figure 5-10a), the transformations are tested and their related information is stored as already described before. But in case a rollback occurs, four main situations can happen:

- The transformation was not successful before, and no transformation is successful ever for the rest of the chunks;
- The transformation was not successful before, but another transformation is successful for the next chunk(s);
- The transformation was successful before, but no transformation is successful ever for the rest of the chunks;
- The transformation was successful before, and another transformation is successful for the next chunk(s).

Figure 5-10: Behavior of the multiversioning system when a rollback occurs.

Therefore, for each handled loop kernel, several different transformations may be successful, but not for all the iterations of the outermost loop. Hence, to be able to cope with this, the mechanism to store the information related to the transformation must be extended. Table 5-4 gathers the information that was already stored –see Table 5-3– and adds the information related to the rollbacks management, highlighted in yellow.

Table 5-4: Information that needs to be stored for each optimizing transformation, evolving from Table 5-2 and Table 5-3.

Item	Description
Loop ID	Unique identifier of the handled kernel
Transformation ID	Unique identifier of the transformation being tested
Execution time	Measured execution time per iteration of the transformation being tested
Profiling results	Results generated during the profiling phase
Prediction model	Prediction model built after the profiling phase
Optimized code	Optimized code generated by the LLVM JIT compiler
Chunk limits	Starting and ending points for which the transformation was successful
Chunk ID	Number of different successful transformations per loop kernel

Specifically, since now more than one transformation per loop can be stored, it is compulsory to save the lower and upper bounds characterizing the outermost loop slice where each optimizing transformation has been successful. Additionally, both

the instrumentation results and the prediction model must depend on those bounds, as they are invalidated when a rollback appears, since they are not correct anymore. Then, after the rollback, if a new valid prediction model is built for the next iterations, and another transformation is proven successful, the information related to the new transformation is stored independently from the previous one. It should be noted that, between two different optimizing transformations, a few iterations of the outermost loop running the original sequential code are always launched, in order to guarantee that the faulty misprediction point generating the rollback is overcome. In a nutshell, the new runtime multiversioning system registers every optimizing transformation whose related optimized code has been successfully run for at least one chunk.

Once the training phase ends, the behavior of the system during the operational phase is as shown in Figure 5-10b: if several transformations have been successfully evaluated, the best one is directly launched for each slice, running a few iterations of the outermost loop with the original serial code between each one. If, after a rollback, the system cannot find a valid transformation, the original code is executed from the moment the last optimization slice finishes until the end of the complete kernel invocation; on the contrary, the next transformation is launched for its specific chunk.

5.5 Code specialization

Up until now, the polyhedral tool selected in this work as a starting point, Apollo, has been extended to become a dynamic system that combines optimizing techniques, such as multiversioning and memoization, to be able to apply the best performing polyhedral transformation for each handled kernel.

However, to increase the robustness of the system, a last aspect must be taken into consideration. As aforementioned, nowadays many applications need to be reactive to changes at runtime to reconfigure themselves to changing environment conditions. This can be the case, for instance, of image and video applications, in which the same processing is applied to a stream of data. However, it is usual for such applications to have to reconfigure themselves if, for example, the image resolution varies.

Since the objective is to allow the new multiversioning system to support this behavior, some modifications must be carried out. The system described until now would not be capable of handling it because the changes in the application have a direct impact on the loop kernels being handled. Following the same example used before, a change in the resolution of the image or video to process would have a direct impact, for example, on the loop bounds of the kernels performing some processing on the image. In turn, a change in the loop bounds of a kernel being handled by the system would end in one of the following situations:

- In the best case scenario, the change in the loop bounds does not imply any change, so the system can continue its execution as before the change occurs.

- However, the change can affect the loop so that the version that was previously evaluated as the best one is not so anymore. For example, it could be possible that a version of a specific tile size stops being the best one if the loop size varies, as that change could provoke the usage of the cache to not be efficient anymore.
- In the worst case scenario, the change in the loop bounds not only implies that the version previously chosen as the best one is not so anymore, but also that the linear functions predicting the memory accesses change too, thus invalidating the prediction model built by Apollo for that kernel in previous invocations.

In two out of the three situations, the framework would not behave accurately or even correctly, since runtime errors could appear due to the usage of an invalid prediction model. Hence, some changes must be performed within the proposed system so it can behave correctly should these situations ever arise, being aware of them and reconfiguring itself accordingly. A simple solution is code specialization, another well-known optimization technique consisting in generating code in which some parameters are initialized as constants to predicted –or already encountered– values, thus generating more efficient code [Consel'96, Grant'99, Poletto'97].

If the parameters that can influence the code structure, the workload distribution and/or the touched memory locations at runtime –which, in turn, are translated to loop bounds, memory accesses, etc– can be somehow identified within the application, then they can be fed to the proposed framework so its runtime manager can associate their state to the information already in use to uniquely identify a specific execution context, thus specializing the code for each value these parameters can take. This is an already known problem, and there are some approaches in the literature where the user can identify at design time which parameters are bound to change at runtime, so the runtime manager can be aware of their state and react accordingly. This is the case of applications implementing the PiSDF MoC, which are the kind of parameterized dynamic dataflow applications targeted by this approach.

As a result, once these parameters have been fed to Apollo –how this is actually achieved from the application will be addressed in Section 6.2–, the runtime manager makes each tested transformation dependent on the value of those parameters for that specific context by storing the set of parameter values together with the information that was already stored for each loop kernel –see Table 5-3–, as highlighted in Table 5-5.

With all this information, Apollo creates an identifier to uniquely characterize each execution context. This identifier is composed by the loop ID, which already identifies the loop kernel that is being handled, and the combination of the specific values of all the dynamic parameters for that execution context. An overview of how the framework works is shown in Figure 5-11. With this change, the framework is finally able to be reactive to possible changes in those parameters: for each loop kernel handled by Apollo and each specific set of parameters, one version is stored, instead of storing just one version per loop kernel. Hence, if there is a change in any dynamic parameter, the execution context changes and different versions are tested during the training phase –Figure 5-11a–, or retrieved during the operational phase –Figure 5-11b.

Table 5-5: Information that needs to be stored for each optimizing transformation, evolving from Table 5-2, Table 5-3 and Table 5-4.

Item	Description
Loop ID	Unique identifier of the handled kernel
Transformation ID	Unique identifier of the transformation being tested
Execution time	Measured execution time per iteration of the transformation being tested
Profiling results	Results generated during the profiling phase
Prediction model	Prediction model built after the profiling phase
Optimized code	Optimized code generated by the LLVM JIT compiler
Chunk limits	Starting and ending points for which the transformation was successful
Chunk ID	Number of different successful transformations per loop kernel
Associated parameters	Specific value of the dynamic parameters for this execution context

Figure 5-11: Complete overview of the dynamic framework including multiversioning, memoization and code specialization: training phase (left) and operational phase (right).

A more detailed view of the implementation is provided in Figure 5-12. As can be seen, each time a loop kernel is launched, the actual values of the dynamic parameters are associated to the execution context, as highlighted with the light blue box.

On the one hand, if, for that combination of parameters, the loop kernel has never been launched before, the process starts from scratch: (i) a small amount of iterations of the outermost loop is executed sequentially to retrieve the profiling information;

Figure 5-12: Diagram of Apollo's dynamic phase with multiversioning, memoization and code specialization.

(ii) with that information, the prediction model is built and Pluto is called to generate a transformation; (iii) once Pluto finishes, the LLVM JIT compiler generates the binary code at runtime; (iv) this code is executed and evaluated through Ehrhart polynomials; and (v) since it is the first time that kernel, with those parameters, has ever been executed, the whole transformation is stored—all the information shown in Table 5-5.

On the other hand, if, for that combination of parameters, the loop kernel has already been launched before—but less than N times, N being a number fixed by the user or automatically initialized within Apollo—the runtime multiversioning system behaves as follows: (i) first, the execution context related to that loop kernel and that combination of parameters is retrieved, namely the instrumentation results and the

prediction model; (ii) with this information, the profiling phase is skipped, and Pluto is directly called to generate a different optimized version than the one(s) previously tested; (iii) the LLVM JIT compiler generates the binary code again; (iv) this code is also executed and evaluated; and (v) this new version is compared to the one that was previously stored: if the new transformation is better than the stored one, the latter is substituted; if not, only the transformation ID is stored to keep track of the transformations already tried for that loop kernel and that combination of parameters.

Finally, if the same loop kernel with the same combination of parameters is revisited more than N times, in successive repetitions of that specific execution context, the best transformation will be directly retrieved and launched, without having to repeat the process described above.

5.6 Summary

During this chapter, the extension of Apollo to become a dynamic framework able to take advantage of the nature of stream-based, data-driven applications to maximize their performance has been detailed. As stated in the introduction of this chapter, the objectives of this improved polyhedral framework are:

- To generate on-the-fly different polyhedral transformations of the handled loop kernels, which are usually relaunched many times;
- To evaluate these different transformations and compare them in terms of performance, so as to select the best one;
- To automatically retrieve and launch this optimizing transformation whenever the same loop kernel is revisited;
- To be reactive to the state of some dynamic parameters, which can change at runtime and affect aspects as the code structure, the workload distribution and/or the touched memory locations, and reconfigure itself accordingly.

As shown in this chapter, to cover these objectives, Apollo has been extended by combining several well-known optimization techniques in the context of code optimization and compilation: multiversioning, memoization and code specialization. In a nutshell, the main contributions of this thesis to the dynamic phase of Apollo are:

- With multiversioning, several different versions of the same loop kernel are generated, executed and evaluated, each including a different polyhedral transformation –or combination of polyhedral transformations–;
- With memoization, the best version for each handled loop kernel, together with its associated prediction model, is stored in memory, so the code can directly be retrieved whenever the same loop kernel is launched again;

- With code specialization, the state of some dynamic parameters is also stored and associated to each loop kernel, creating specific execution contexts for each combination of these parameters, so all the process explained above is specialized for each loop kernel and each combination of parameters.
- Additionally, to provide a robust framework, a new figure of merit, based on Ehrhart polynomials, to accurately evaluate each version in terms of performance has been included within Apollo.

With all the new features, Apollo has evolved to become a runtime multiversioning system which is now prepared to be integrated with a dataflow framework to fully exploit its optimization possibilities, which is one of the main objectives of this thesis.

Similar to what has been done in this chapter, Chapter 6 will present all the modifications needed to be performed within the dataflow tool so as to fully integrate it with Apollo. Nonetheless, it will also cover some modifications that have still to be done within Apollo, and which have not been addressed here because their objective is not to modify the behavior of the tool from the polyhedral point of view, but to ease the integration of both tools, in some cases, or more importantly, to make it possible –as, for example, how to feed the dynamic parameters from the application to Apollo.

6

PISDF MOC EXTENSION THROUGH POLYHEDRAL TRANSFORMATIONS

Chapter 5 detailed the extension of a polyhedral framework such as Apollo to turn it into a runtime speculative system that dynamically explores different polyhedral optimizations for compute-intensive kernels, selects the best one and applies it in the different invocations of the same kernel, combining several well-known optimization techniques, such as multiversioning, memoization and code specialization.

However, some additional steps must be given in order to fully integrate this tool with a dataflow framework to conceive the extended Y-chart, dataflow-based design methodology presented in Figure 1-2. For instance, last chapter concluded by stating the need for some communication channels through which the actual values of the dynamic parameters at each invocation of a given kernel can be fed to Apollo. Nonetheless, this is not the only issue that must be addressed so as to fully automate the integration of both domains. Up until now, when addressing the changes needed to extend Apollo, the loop kernel targeted to be optimized was referenced many times, but its connection with a dataflow application remains unclear.

Hence, the extended Y-chart design methodology proposed in Figure 1-2 must be modified so as to fully integrate dataflow and polyhedral worlds. These modifications are divided into three different concerns, as shown in Figure 6-1:

- **Polyhedral-dataflow interface ❶:** The examples used before to show the work that must be finished to complete the proposed design methodology fully justify the need for an interface to communicate and connect dataflow and polyhedral domains. This part will detail the development of this interface, focusing on how it solves the open issues to establish a full communication between the two parts.
- **Dataflow preparation ❷:** As advanced at the end of the previous chapter, to achieve this full integration and exploit completely the nature of dataflow applications, some final changes need to be performed within Apollo. In this part, these changes will be described in detail, highlighting why they must be included in the framework and what are the advantages they provide.
- **Polyhedral preparation ❸:** This category gathers all the changes needed to be addressed from the point of view of the dataflow framework so as to prepare the code of the application to be fed to the polyhedral framework, in this case Apollo.

As a result, this chapter presents all the modifications needed to interconnect the extended version of Apollo presented in last chapter with PREESM and its

dynamic counterpart, SPiDER, whose selection among the state-of-the-art dataflow environments was justified in Chapter 3. In that regard, the rest of the chapter is organized as follows: first, the interface needed to communicate both worlds will be detailed in Section 6.1; afterwards, the changes needed to be added both in polyhedral and dataflow domains so as to fully integrate and make use of the interface will be presented in Section 6.2 and Section 6.3, respectively. Finally, a summary of all the contributions presented in this chapter will be gathered in Section 6.4.

Figure 6-1: Overview of the specific tasks envisioned to embed the polyhedral domain in the Y-chart, dataflow-based design methodology.

6.1 Polyhedral-dataflow interface

As justified in Chapter 3, PREESM is the dataflow framework chosen to implement the extended Y-chart design methodology proposed in this work, and was described in detail in Chapter 4. As stated there, PREESM is based on PiSDF MoC, which consists of modeling applications as a directed graph –or *network*– with a set of computational units, called *actors*, that interchange information –*tokens*– through FIFO queues. PiSDF MoC presents two specific characteristics: first, it is hierarchical,

so each actor of the network can contain, in turn, a whole new network; furthermore, it is parameterized, so the behavior of the whole application depends on the value of certain parameters, which may in turn influence the structure of the network, either at compile time or at runtime, through PREESM's runtime manager, SPiDER.

Chapter 5 described the modifications carried out within a polyhedral tool so as to prepare it to provide the best performance achievable from a dataflow application. However, this chapter left some important questions unanswered, namely how to connect such a tool with a dataflow framework like PREESM. One of the easiest solutions to connect both frameworks is to design an API through which both of them can interchange the information needed to make them work together.

However, before getting to the design of the API, the terminology used in Chapter 5 must be clearly mapped to its counterpart in the dataflow domain, so the common links between the worlds can be identified. In that regard, there are two main concepts intensively used throughout Chapter 5: *loop kernel* and *parameters*.

Loop kernel is one of the most used concepts in Chapter 5, and also in Section 4.1, where the polyhedral model was introduced. In a nutshell, it is a mathematical framework aiming at analyzing and transforming loop nests so as to enable automatic loop optimization and parallelization. Hence, the target of all the tools based on this model are loop nests, usually compute-intensive, which take a considerable amount of time to be executed and, thus, are susceptible to being optimized and parallelized. Hence, from the polyhedral domain perspective, the loop kernel is clearly identified. However, from the dataflow domain point of view, this loop kernel is not yet defined. As aforementioned, a PREESM application, specified following the PiSDF MoC, is divided into a set of actors that perform computations, these computations being enclosed within a function that is internally associated to each actor. These functions must comply with some rules, namely that all the input and output ports are represented as parameters of the function, but other than that, within the function, one can come across a wide variety of codes. Consequently, each actor can contain zero, one or more loop nests susceptible of being targeted by the polyhedral tool, or not. Additionally, as described in Chapter 4, PREESM offers the possibility of parallelizing actors in such a way that, if an actor has available at its input ports more tokens than it consumes, several instances of the same actor will be created in parallel until this production-consumption rate is balanced. These instances have the same function associated to them and, therefore, perform the same computation, but over different sets of data, which means that the same loop kernel can be executed from different actors.

To sum up, from the point of view of the dataflow domain, a *loop kernel* is a loop nest, within an actor, that the developer wishes to accelerate. This statement implies, in turn, two important characteristics of loop kernels within dataflow applications that must be taken into account when connecting the two domains:

- The number of loop kernels within an actor can be more than one.
- The same loop kernel can be repeated in different actor instances.

Hence, to make Apollo able to optimize any loop nest within a dataflow application, the latter must send to the former, at least, a unique identifier of the instance of the actor, so it can be linked to the kernel identifier already included in Apollo to singularly denote each kernel targeted by the polyhedral tool, as described in Section 5.1.

On the other hand, *parameters* have already been introduced while defining the PiSDF MoC. As introduced in Section 4.2, parameters within PREESM and SPiDER are directly identified in the application network through blue triangles and, in the case of the latter, through blue triangles with a white circle if they present a dynamic behavior. These parameters directly affect the structure and shape of the network, allowing users to modify the dataflow specification of an application easily, keeping them from having to modify the inner code associated to the actors. Additionally, in the case of SPiDER, these parameters may change at runtime, dynamically modifying the shape and structure of the application network.

As a result, when thinking about linking these dataflow specifications with the polyhedral framework proposed in Chapter 5, it is of vital importance to be able to communicate these parameters to the polyhedral framework, so it can register them and include them within the runtime multiversioning system.

All these reasons fully justify the design and implementation of an API to communicate both frameworks. To sum up, the dataflow system needs the API to inform the polyhedral framework of, at least: (i) an identifier of the instance of the actor that is being executed, so loop kernels can be singularly identified; and (ii) the different values the dynamic parameters take during the execution of the dataflow network. Yet, there is still more information to be shared between both worlds, and that can be useful to simplify both the integration of dataflow and polyhedral worlds and the usage of the unified framework by the developers: for instance, users can directly configure how many transformations per loop kernel they want to evaluate.

Consequently, to take into consideration all these features, the API presented in this section has been organized in three different groups, according to their functionality: *initialization*, *kernel* and *others*, as shown in Table 6-1. This table also gathers the name of the functions within each category, together with a brief description.

- **Initialization:** This group gathers the functions added to the API related to the initial configuration any dataflow application using Apollo should perform:
 - *Enable / Disable multiversioning:* These two functions allow the user to turn on or off, respectively, the multiversioning mechanism developed within Apollo and thoroughly described in Chapter 5. With these functions, users can automatically activate or deactivate the new functionalities added to Apollo, granting thus its usage as an extension of the tool, and allowing users –both from dataflow and polyhedral domains– to use either the enhanced version of Apollo or the original one.
 - *Enable / Disable parallel:* These two functions give the users the power over Apollo’s parallelization potential. As described in Chapter 5, this tool

Table 6-1: Polyhedral-dataflow API.

Functionality	Function	Description
<i>Initialization</i>	apolloEnableMultiversioning()	Activate multiversioning within Apollo
	apolloDisableMultiversioning()	Deactivate multiversioning within Apollo
	apolloEnableParallel()	Activate parallelization within Apollo
	apolloDisableParallel()	Deactivate parallelization within Apollo
	apolloSetTxnToTry()	Set the number of transformations to try
	apolloSetMode()	Configure how Apollo will work
<i>Kernel</i>	apolloAddActorConfig()	Send the characterization of an actor
	apolloAddDynamicParams()	Send the value of the dynamic parameters
	apolloAddJobIdx()	Send the identifier of the dynamic actor
<i>Others</i>	disableApollo()	Turn off Apollo
	apolloGetEstimatedTime()	Estimate the time of the optimized kernel
	apolloStopTxSelection()	Stop training phase within Apollo

can both optimize the code –by improving, for instance, data locality– and parallelize it –by adding OpenMP directives–. However, as described in Section 4.2, PREESM and SPiDER also provide a mechanism to parallelize applications. Combining both of them is a problematic issue, since the dataflow tools should be aware of the number of threads Apollo is creating to handle them together with the threads they already create, since PREESM uses its own scheduler, who is unaware of the presence of Apollo. As a result, these two functions have been created to handle this issue, giving the possibility to let Apollo behave strictly as an optimizer, disabling its parallelization potential. Section 6.2 will analyze how the parallelization capabilities of PREESM affect Apollo, and how it has been addressed.

- *Number of transformations to try:* As also described in Chapter 5, the multiversioning mechanism evaluates different transformations for each kernel and keeps the best performing one. Although the number of transformations to test is initialized to a default value within Apollo, it can be easily modifiable. With this function, the user can now decide the size of the set of transformations to test. It should be noted that, if the user configures this number to N , only the first N transformations from the set created by default within Apollo will be tested. To fully change the set of transformations to test, the user must modify it directly within Apollo. The reason why the set is not modifiable from the API is that, in this version, the API only allows modifying those aspects of Apollo that may cause conflicts from the dataflow point of view, such as the parallelism or the working modes; nonetheless, it can be easily included in the API in future versions.
- *Mode:* As hinted at the end of Chapter 5, there are still some modifications to be performed within Apollo to fully exploit the nature of dataflow systems and maximize their performance. One of these modifications is related to the mechanism Apollo has to mask the overhead added due

to the computation of the prediction model and the invocation of Pluto at runtime, which roughly consists of launching a background thread executing the original code. This mechanism can be changed to comply with dataflow MoCs. Section 6.2 will address both these modifications and how users can apply them through the function added to the API.

- **Kernel:** This group contains the functions needed to send all the information required to uniquely identify a loop kernel to Apollo, so the optimizations can be applied to any loop kernel within the application:
 - *Add actor configuration:* The objective of this function is to fully characterize the actor within Apollo. To do so, this function takes three input parameters: (i) the name of the instance of the actor, so as to uniquely identify each instance existing at each invocation of the network; (ii) a flag to indicate if the actor is static or dynamic –i.e., if the network contains dynamic parameters–; (iii) an identifier of the thread that is executing the actor with the loop kernel, so as to keep track of where the actors are executed and be able to map the instance of the actor with the loop kernel, whose identifier is gathered directly within Apollo. The details of this function will be further explained in Section 6.2.
 - *Add dynamic parameters:* This function sends to Apollo the current values of the dynamic parameters of the dataflow network, so they can be taken into consideration in the multiversioning system.
 - *Add job identification:* This function is a wrapper of the first one, and is going to be used to configure actors in SPiDER. As the rest of them, it will be further detailed in Section 6.2.
- **Others:** Finally, this group gathers the rest of the functions that do not fit in any of the previous groups, but which are still useful for the integration of dataflow and polyhedral domains:
 - *Disable Apollo:* It disables the usage of Apollo. It has been added as a safeguard, in case the user accidentally enables Apollo but it is not correctly configured. It will be further addressed in Section 6.3.
 - *Stop training phase:* This function has been added as an example of the additional possibilities offered by extending the dataflow domain through polyhedral transformations. This function allows users to stop the training phase of Apollo before all the training set has been tested. With this function, the control of the multiversioning system is handed to the dataflow domain, instead of remaining within the polyhedral one.
 - *Estimated time:* It can be used to estimate the execution time of a given transformation for the whole loop kernel –not just for a few iterations–. Coupled with the previous one, this function can be used, for example, to stop the training phase whenever a transformation reaches a certain

performance threshold, without waiting until all the transformations in the set have been evaluated and, thus, reducing the overhead of the training phase. Although it is just an example, it helps in showing that, thanks to the API, adding functionalities to the new framework is relatively easy.

As aforementioned, one of the main advantages of having implemented this API is that it considerably simplifies the process of adding new functionalities to the new framework; likewise, it shifts the control of the multiversioning system from the polyhedral domain to the dataflow one, making Apollo depend on the dataflow framework. Yet, as Apollo was initially devised to be used as a standalone framework, this API adds some safeguards to maintain its standalone usage.

The following sections will get deeper into the details of some of the aforementioned functions, outlining how all the pieces fit together.

6.2 Dataflow preparation

Section 6.1 has introduced the API designed to combine the dataflow and polyhedral domains. Now, this section is going to detail the impact of the changes laid out by the API within Apollo. Following the description of the functions conforming the API, their impact within Apollo can be categorized in two main groups: (i) the changes arisen as a result of the parallelization capabilities of PREESM, studying how this affects Apollo's functioning –Subsection 6.2.1–, and (ii) the changes needed to address in order to make the multiversioning mechanism fully work with the information sent from the dataflow framework to fully characterize each loop kernel –Subsection 6.2.2.

6.2.1 Combining parallelization capabilities

Regarding the parallelization possibilities offered by PREESM, Section 6.1 already advanced that it affects the functioning of Apollo, and thus it should be taken into consideration in order to build an enhanced framework which takes advantage of the optimization capabilities offered by both tools in a robust way. To do this, first the impact the parallelization capabilities of PREESM have over Apollo is going to be analyzed, so then a solution for each issue arisen can be found.

As described in Section 4.2, PREESM presents a mechanism to ease the parallelization of applications, which is twofold: first, it can map and schedule different actors into different threads so they can be executed in parallel if they do not have data dependences; secondly, it can exploit the actors' inner data parallelism by creating several instances of actors whose processing can be applied simultaneously to different sets of data. In both cases, PREESM relies on creating a thread per physical PE available in the target architecture, and then using its own mapper-scheduler to determine where and when each actor is going to be executed.

From the point of view of Apollo, this behavior can create several issues, jeopardizing the full integration of both tools to a greater or lesser extent.

The first issue is related with Apollo's concurrency. As aforementioned, loop kernels are inside the functions containing the processing associated to the dataflow actors. Considering too the parallelization mechanism described before, it means two actors, both with a loop kernel, can be executed simultaneously in different threads. From the point of view of the dataflow framework, this does not create any issue, since this framework was devised to support parallelization from the beginning. However, the same does not apply to the polyhedral framework, Apollo. This tool was originally devised as a standalone tool, so its primary use was intended to be in sequential programs the user wishes to automatically parallelize, thus without any concurrency before Apollo's own creation of threads. Unfortunately, when integrating this tool with the dataflow framework, Apollo is now managed from the dataflow side, and its original use does not hold anymore. Hence, the tool has been revised to detect any concurrency issues that could prevent both tools from working together successfully. This analysis concluded that both Apollo runtime and some of the libraries it uses to perform the dependence analysis and the polyhedral transformations –namely, Chunky Analyzer for Dependences in Loops (Candl) ¹, Parametric Integer Programming Library (PipLib) [Feautrier'88] and Pluto– presented several concurrency issues, mainly due to the usage of global variables. Consequently, all the tools have been made thread-safe, and thus can be used in a concurrent fashion.

The second issue is related with combining the parallelization capabilities of both domains. As already advanced in Section 6.1, both PREESM and SPiDER use their own schedulers and do not consider the possibility of other elements using the same resources and generating additional threads, as is the case of Apollo. However, this situation will happen whenever Apollo tries a transformation, unless otherwise specified by the user. In that case, none of the frameworks working together would be aware of the presence of the other, so they would assume that they can use all the resources available in the target architecture. For instance, if the target architecture contains 4 PEs, the situation would be the following: first, PREESM would perform its own mapping and scheduling, assuming that the 4 PEs are available, by creating 4 threads; then, during the execution, whenever a loop kernel is encountered, Apollo's runtime manager would be launched, who in turn would also consider that it can use the 4 PEs of the target architecture to parallelize the loop nest, creating additional threads through OpenMP. As a result, during the execution of the applications, there would be moments in which a maximum of 16 threads could be created, but only 4 physical PEs to execute them. This could lead to a performance loss, which is opposite to the initial objective of combining both frameworks.

Therefore, the solution proposed in this work is to allow users to configure the parallelization to use, either the one of PREESM or the one of Apollo. Thanks to the new functions in the API –`apolloEnableParallel()` and `apolloDisableParallel()`–, the parallelization inside Apollo can be easily deactivated, thus turning Apollo into an

¹http://icps.u-strasbg.fr/people/bastoul/public_html/development/candl/

optimizer, but not a parallelizer. On the contrary, if users want to use the parallelization within Apollo, PREESM should be configured using a 1-core scenario, that is, simulating that only one PE is available in the target architecture, and thus forcing PREESM to generate sequential code. This workaround offers the possibility of comparing both parallelization capabilities, studying them individually so as to be able to propose a solution to fully integrate them in the future, such as a heterogeneous and hierarchical scheduler that dynamically balances these capabilities depending on the application and the target architecture to exploit both inter- and intra-actor parallelization. This future line will be further described in Subsection 9.4.2.

Nonetheless, enabling one framework or another to parallelize code is not the only issue arising from combining these two frameworks devised for parallelizing applications. Even if Apollo is configured to behave just as an optimizer, there is a moment in its runtime execution in which an additional thread appears: as described in Section 4.1, this framework creates a background thread that executes the original code of the loop kernel sequentially while the main thread is computing the prediction model and calling to Pluto to generate the transformation. The objective of this mechanism is twofold: (i) to counterbalance the overhead introduced by these phases; and (ii) to counteract Apollo's overhead in such a way that, if the sequential thread finishes before the prediction model or Pluto invocation, Apollo's execution is immediately aborted, and the results of the sequential thread directly provided to the user. With this mechanism, the loop kernel is guaranteed to not be considerably slowed down by Apollo when the handled loop kernel is too small regarding its workload.

Considering the issues an extra thread may provoke, this is not a desirable situation when integrating Apollo with a dataflow framework such as PREESM, so this mechanism is going to be revisited. Additionally, reconsidering this mechanism opens in turn new possibilities to further exploit the opportunities offered by the combination of a dataflow framework with a runtime polyhedral parallelizer as Apollo. As aforementioned, dataflow applications are running on a loop; in this context, a new compromise may be reached, considering the possibility of letting Apollo spend some extra time in generating the optimizing transformation. Hypothetically, this time will be canceled out by the speedup provided in the operational phase. This functionality can be implemented within Apollo as a timeout, configured by the user and only activated if the sequential thread finishes before the optimized code has been generated. Theoretically, this extension would benefit small loop kernels, as they could not be handled profitably by Apollo before.

Consequently, several approaches can be followed to modify this mechanism. So as to provide users with as much flexibility as possible, several modes of functioning have been implemented within Apollo to control the behavior of the mechanism to mask the overhead. There are five different modes, summarized in Table 6-2, which are basically defined by two key features: whether or not the sequential thread is launched, and the value of the timeout for generating the optimizing transformation.

- **Mode 1:** Launches the sequential thread and aborts Apollo's execution if this thread ends before the transformation is computed (Apollo's original behavior).

- **Mode 2:** Launches the sequential thread and establishes a configurable timeout of N times the time measured for the sequential thread.
- **Mode 3:** Equal to mode 2, but establishing a configurable timeout of Y seconds.
- **Mode 4:** Equal to mode 3, but without launching the sequential thread.
- **Mode 5:** This mode has been added as a safeguard. It does not follow the rules of the rest, it is just used to make Apollo execute only the original code, without trying any transformation. The objective of this mode is to be a safeguard with respect to the dataflow framework, and execute the code sequentially if Apollo is not correctly configured or initialized. It will be further addressed in Section 6.3.

Table 6-2: Summary of the different modes defined in Apollo.

Mode	Sequential thread	Timeout	Function
Mode 1	✓	-	Apollo original behavior
Mode 2	✓	$N \times$	$N \times$ timeout
Mode 3	✓	Y seconds	Y seconds timeout
Mode 4	✗	Y seconds	PREESM specific usage
Mode 5	✗	-	Safeguard mechanism

As can be observed, mode 4 does not launch the sequential thread: this mode has been thought for being used from PREESM when PREESM is using its parallelization capabilities; if the timeout expires, Apollo will launch the original code for the rest of the chunk as happened before, but it will start from the iteration in which the previous phase –the instrumentation– ended, and hence there will not be any masking of the overhead. However, if PREESM is parallelizing actors, this is the best solution, since presumably the speedup obtained in future invocations will cancel out this overhead.

However, if the user wants the parallelization capabilities handed over to Apollo, then modes 2 and 3 can be used, as they launch the sequential thread and, thus, allow the application of the overhead masking mechanism, their only difference relying upon the kind of timeout they apply: the former has a relative timeout, while the latter presents an absolute one. As already pointed out in Section 6.1, this configuration can be chosen through `apolloSetMode()` function of the API.

6.2.2 Feeding dataflow information to the polyhedral domain

In Section 6.1, the information the polyhedral framework needs to uniquely characterize a dataflow actor has been described, and the function added to the API to that purpose, `apolloAddActorConfig()`, has been presented. To close this section, the only aspect left is to explain how this information affects Apollo, and thus how it is integrated to make the multiversioning system work with dataflow applications.

As described in the previous section, there are some pieces of information required to fully characterize each loop kernel within a dataflow application, namely: (i) an identifier of the loop kernel being optimized; (ii) an identifier of the actor in which the loop kernel is embedded; (iii) an identifier of the thread running that actor; and (iv) a flag indicating if the network embedding that actor can present dynamic parameters or not –i.e., if it comes from SPiDER or PREESM, respectively–. The first one was directly retrieved thanks to Apollo’s static phase, which directly generates a unique identifier of the pragma linked to the loop kernel that is triggering Apollo’s runtime phase. On the contrary, the other three parameters come from the dataflow domain, and are passed to Apollo through the `apolloAddActorConfig()` function.

Once inside Apollo’s runtime phase, this information needs to be integrated within the runtime multiversioning system described in Chapter 5. But, before doing so, first it is important for Apollo to correctly link the information that is coming from the dataflow system with the data it already contains; in other words, Apollo needs to associate the identifier of the loop kernel it already has with the pieces of data coming from the dataflow system through the API. This is a matter of special relevance, since, as aforementioned, PREESM can create several instances of the same actor, and all of them execute the same code but on different data, as depicted in Figure 6-2, where two instances of the same actor, *Sobel0* and *Sobel1*, apply a filter to their corresponding half of the image. When translated to the polyhedral domain, this means that there can be several loop kernels with the same identifier, but coming from different actor instances, and, even worse, using input data located at different memory addresses. For that reason, even though the instances execute the same loop kernel, their associated prediction models computed by Apollo may vary, and thus each instance should be treated separately by the multiversioning system.

Figure 6-2: Example of two instances of the same actor applying the same processing to different input data simultaneously.

Additionally, from the point of view of the dataflow system, it is not possible to retrieve the identifier associated to each loop kernel so as to link it with the identifier of the actor, so a new mechanism to link these two pieces of information needs to be devised, and that is the reason why an identifier of the thread that is executing each actor is passed to Apollo.

Figure 6-3 provides a toy example to fully illustrate this issue, which continues the example shown in Figure 6-2. In this example, two instances of actor *Sobel*—*Sobel0* and *Sobel1*—are created to divide its processing into two threads, *Th0* and *Th1*, which will then be executed by different PEs of the target architecture.

Figure 6-3: Overview of the mechanism designed to link dataflow and polyhedral information.

If the user wants to use Apollo to further improve the performance of *Sobel*, Apollo's dedicated pragma must be included in the code associated to the actor—*sobel.c*—. When the application is launched, eventually *Sobel0* and *Sobel1* will be executed in parallel: at that moment, each actor instance will ❶ write the information passed through `apolloAddActorConfig()` to a non-volatile memory region inside Apollo—(a) for *Sobel0* and (b) for *Sobel1*— and ❷ invoke Apollo's runtime system on its own when coming across the pragma. In that moment, Apollo's dynamic phase is launched twice, one per thread: when each call happens, Apollo's runtime retrieves the identifier of the pragma—i.e., the identifier of the loop kernel—, but, since this identifier is the same for both actor instances, it does not know which parts of the information written in its non-volatile area are associated to each instance. This can be solved if the identifier of the thread is used to pinpoint the pieces of data coming from the dataflow world. As can be seen in Figure 6-3, Apollo's runtime is always invoked from the thread executing its corresponding invoker actor, so, if the parameters that are being passed to the polyhedral domain are linked to the identifier of the thread, Apollo can use this identifier to retrieve the correct information and correctly launch the different runtime phases presented in Section 4.1 and repeated in Figure 6-3.

Consequently, with this mechanism, Apollo can now associate the information coming from the dataflow world with the one it already stored for each loop kernel. In Chapter 5, Table 5-5 detailed the data stored for each loop kernel for the multiversioning system to work in a robust and efficient way. These data need to be completed now with this extra information, as highlighted in Table 6-3, so as to extend this robustness and efficiency to the integration with the dataflow system.

Table 6-3: Information that needs to be stored for each optimizing transformation, evolving from Table 5-5 and including the information coming from the dataflow domain.

Item	Description
Loop ID	Unique identifier of the handled kernel
Transformation ID	Unique identifier of the transformation being tested
Execution time	Measured execution time per iteration of the transformation being tested
Profiling results	Results generated during the profiling phase
Prediction model	Prediction model built after the profiling phase
Optimized code	Optimized code generated by the LLVM JIT compiler
Chunk limits	Starting and ending points for which the transformation was successful
Chunk ID	Number of different successful transformations per loop kernel
Associated parameters	Specific value of the dynamic parameters for this execution context
Actor ID	Unique identifier of the actor containing the handled loop kernel
Static / dynamic flag	Flag indicating if the dataflow application can have dynamic parameters

As detailed in Chapter 5, with the information shown in Table 5-5, Apollo builds an identifier that uniquely characterizes each execution context. It is composed by the loop ID, which denotes the loop kernel being handled, and the combination of the specific values of all the dynamic parameters for that execution context.

With the new information, the only step left is to modify this unique identifier to make it depend on every piece of information needed to singularly denote a loop kernel embedded in a dataflow application. This identifier is built as a combination of:

- **Loop ID:** Unique identifier of the loop kernel.
- **Actor ID:** Unique identifier of the actor containing the loop kernel. This identifier is going to vary depending on the flag indicating whether there are dynamic parameters or not: if no dynamic parameters are detected, this identifier is going to be the name of the instance of the actor; otherwise, this identifier is going to be an identifier of the job executing the corresponding instance of the actor, which is passed through `apolloAddJobIdx()` function of the API.
- **Combination of dynamic parameters:** Combination of the specific values of the dynamic parameters for which the loop kernel is being executed. If the network is dynamic –i.e., it comes from SPiDER– but no dynamic parameters are detected, a default value is passed as specific value of the dynamic parameters.

As for the combination of dynamic parameters, this information is directly written in the `apolloAddDynamicParams()` function of the API. It should be noted that the thread identifier is not used to build this unique identifier, since its usage is limited to associating in each moment the information coming from the dataflow world with the data already stored within Apollo.

With this last modification, the runtime multiversioning system depicted in Chapter 5 can work as described there, closing the integration with the dataflow

system. With all these modifications, Apollo is completely extended to be used both as a standalone tool and as a part of an enhanced dataflow framework, providing new features oriented to exploiting the nature of iterative applications while, at the same time, maintaining its original functioning for any other kind of application.

To close this chapter, next section details the changes included within the chosen dataflow framework, PREESM –and SPiDER–, to automate the usage of Apollo as a framework to further optimize compute-intensive dataflow applications.

6.3 Polyhedral preparation

Until now, this chapter has dealt with how to communicate dataflow and polyhedral domains –Section 6.1– and how the polyhedral domain should be adapted to work with an application coming from the dataflow domain and to fully exploit the optimization possibilities appearing as a result of applying polyhedral transformations to iterative applications –Section 6.2–. In other words, Section 6.1 presented the API designed to communicate both domains, and Section 6.2 outlined how to use it within the polyhedral domain, that is, how this domain has evolved to include the functionalities of the API. Now, there are only two aspects left: (i) describing how the API has to be used from the point of view of a dataflow application, so as to configure it to be optimized through Apollo; and (ii) depicting how this process has been automated within PREESM and SPiDER so as to minimize the effort of developers to use this new framework.

The changes that should be added within dataflow applications to make them work with a polyhedral framework as Apollo can be summarized into three groups, following a top-down approach, as shown in Figure 6-4, where the modifications are highlighted in dark purple: framework-level, application-level and actor-level changes.

Figure 6-4: Adaptation of dataflow applications to be compliant with Apollo.

- **Framework-level changes:** This category corresponds with the changes that are needed from the point of view of the whole dataflow framework to make it work with a polyhedral tool like Apollo. As described in Section 4.1, Apollo is composed of two different parts: a static one, responsible for compiling the application and extracting some relevant information, and the dynamic one, which mainly consists of a runtime manager orchestrating the execution. Consequently, to make Apollo work combined with any kind of framework, the first step is to change the compiling toolchain of that framework from whichever the original one is to Apollo's own compiler, which is based on LLVM.
- **Application-level changes:** Once the compiler is changed to that of Apollo, each dataflow application must be configured to use Apollo either to parallelize or to optimize. This group gathers the changes needed within each application as a whole, and which will be maintained for the whole execution. As described in Section 6.1, some of the functions included in the API were designed to configure the application, namely `apolloSetTxnToTry()`, `apolloSetMode()`, `apolloEnableMultiversioning()/apolloDisableMultiversioning()` and `apolloEnableParallel()/apolloDisableParallel()`. As this configuration must be common for any loop kernel in the application, if users want to further optimize a dataflow application through Apollo, these functions must be called at the beginning of the program's execution, before any actor is launched.
- **Actor-level changes:** Finally, once both the framework and the application are configured to work with Apollo, the only aspect left is to configure each actor separately, so the possible loop kernels inside them can be fully characterized from the point of view of Apollo. As described in previous sections, the functions added to the API to handle that aspect are `apolloAddActorConfig()` for PREESM, and `apolloAddJobIdx()` and `apolloAddDynamicParams()` for SPiDER. This configuration is automatically added for every actor in the application, since, at the actor-level, the actors invoking Apollo cannot be discerned from those not doing it. In Figure 6-4, however, this behavior is simplified for illustration purposes, showing only the configuration of those actors invoking Apollo, which are denoted with a #.

So, to sum up, if users want their dataflow applications modeled in PREESM or SPiDER and further optimized through Apollo, they have to follow the next steps:

1. Select Apollo's compiler as the one used to compile the dataflow application.
2. Add the initialization of Apollo at the beginning of the application, before any actor is executed. To simplify this process, a new function has been added to the API –`initApolloForDataflow()`–. It does not include any new functionalities, but gathers all the necessary steps to configure the application:
 - (a) *Number of transformations to try:* By default, this value is initialized to 15, but it can be easily modified by the user.

- (b) *Enable multiversioning*: By default, the multiversioning mechanism is activated, since hypothetically it is going to provide better results than the previous version of Apollo, which did not include this mechanism.
 - (c) *Disable parallel*: Due to all the issues arisen as a consequence of the parallelization capabilities of PREESM and SPiDER, which were explained in Section 6.2, by default the parallelization capabilities of Apollo will be disabled, but they can easily be enabled by users if they decide so.
 - (d) *Set mode*: Following the same philosophy than with the previous function, to avoid the issues arisen from combining the parallelization capabilities of both tools, mode 3 is going to be chosen as default, since it does not launch the sequential thread; as for the value of the timeout, since it is an absolute value, it depends on the application, so the user needs to configure it.
3. Add the specific configuration of each actor through `apolloAddActorConfig()` –in the case of PREESM– and `apolloAddJobIdx()` –in the case of SPiDER, since this function is a wrapper of the former to substitute the actor identifier–. This function must be called immediately before the execution of the related actor, so Apollo has all the information when its runtime manager is triggered.
 4. Add the current value of the dynamic parameters, if any, at the beginning of each iteration, when the eventual changes in the dynamic parameters are detected. This function –`apolloAddDynamicParams()`– is only used in SPiDER, since PREESM only deals with static parameters.
 5. Add the dedicated pragma, `pragma_apollo_dcop`, surrounding the kernels to be optimized. This instruction must be added in the code of each actor, which is not generated automatically by PREESM, but manually created by the user.

Finally, to close the description of the enhanced methodology proposed to accelerate dataflow applications, this chapter ends with a summary of the changes performed within both PREESM and SPiDER to fully automate the process described before and ease the usage of the new framework from the users' point of view.

Regarding PREESM, as outlined before, only two functions need to be added: the initialization function, at the beginning of the execution, and the function with the configuration of each actor, just before executing the actors. As described in Section 4.2, PREESM generates a separate file for each physical PE the user wants to use and containing the code associated to its thread, with the succession of actors this thread is going to execute. Additionally, to orchestrate the execution and create the threads, a main file is also created. Consequently, the selected solution has been adding the call to `initApolloForDataflow()` function in the main file, since this code is common for all the resources, and adding the call to `apolloAddActorConfig()` before each actor is executed, inside the code associated to each PE. As all these files are automatically generated by the static backend of PREESM, it has been decided to include the call to these functions within the code generation, so as to

minimize the user's effort. Additionally, the import of the file containing the API is automatically added to another file automatically generated by PREESM. To do so, the *Code Generation* step of PREESM workflow –which is shown in Figure 4-13– has been modified to add a new input flag, *Apollo*, which must be set to *true* by users if they want to further optimize the application with Apollo. When activated, this flag enables the insertion of the aforementioned instructions in the positions mentioned before.

Figure 6-5: PREESM modified workflow.

To provide the combination of tools with robustness, a mechanism has been added to cancel the effect of Apollo in case it is not properly configured. For example, if the user forgets to remove the dedicated pragma but deactivates the flag in PREESM workflow, the calls to the functions initializing the multiversioning system and gathering the information that must be sent to Apollo are not inserted, and thus Apollo will behave unpredictably. For those situations, a specific function, `disableApollo()`, has been added to the API as a safeguard, as mentioned in Section 6.1. This function disables the multiversioning system and configures Apollo with mode 5, which only executes the original code sequentially, without trying any optimizing transformation.

As a last step, the process of changing the compiler from the one PREESM originally used to the one developed for Apollo has also been automated. Now, users only need to activate a flag in the CMake file to indicate that Apollo is used, and the compiling toolchain will automatically detect if Apollo is installed in the platform or not; if not, its usage will be automatically disabled through the function described above.

To fully illustrate how the combination of Apollo and PREESM framework works, a tutorial describing this process has been uploaded to PREESM official website². It can be used as a starting point by those developers that wish to test the advantages offered by the combination of these tools. This tutorial guides users into the proper usage of Apollo within a PREESM application.

Last, but not least, regarding SPiDER, the methodology followed is very similar to

²<https://preesm.github.io/tutos/apollo>

the one applied in PREESM, but with some slight differences. SPiDER workflow was depicted in Figure 4-17 and repeated in Figure 6-6; as can be seen, this workflow is much more reduced than that of PREESM. Consequently, as a first step, a flag indicating whether the user wants to use Apollo or not has been added –highlighted in red in Figure 6-6–, similarly to what has been done with PREESM. This flag is passed to SPiDER to let it know that the user wants to use Apollo.

Figure 6-6: SPiDER modified workflow.

However, both the mapping and scheduling within SPiDER are performed at runtime, since they can vary depending on the value of the dynamic parameters. As a result, some crucial information, as the identifier of the actor, cannot be known from the point of view of the code generation; consequently, the changes needed to address to make SPiDER work properly with Apollo have been directly included within SPiDER's runtime manager, which is implemented as a shared library. Specifically, the initialization process has been embedded into SPiDER's own initialization routine, while the function adding the actor configuration, `apolloAddJobIdx()`, is included in each LRT –i.e., the lightweight runtimes executing the actors–, since until that moment it is not possible to obtain an identifier of the instance of the actor that is going to be run. Finally, the call to the function sending the value of the dynamic parameters to Apollo is added in the routine responsible for detecting a change in them.

Once all these modifications have been included within SPiDER's runtime system, next step is to compile SPiDER's shared library with Apollo. As with PREESM, to ease the usage of Apollo, an automatic process has been added to the compiling toolchain to automatically detect if Apollo is installed in that platform or not, so as to disable its usage –through `disableApollo()`– in case it is not detected in the platform.

As a summary, in order for SPiDER to work properly with Apollo, the user needs to activate the flag in SPiDER's workflow and have Apollo installed in the target machine; in any other case, Apollo will trigger the safeguard described before.

This section finishes the chapter, as well as the integration of polyhedral and dataflow frameworks. These last two chapters have depicted the enhanced metho-

dology proposed in this document, together with an example of its implementation with specific state-of-the-art frameworks, which in turn have had to be modified and adapted. The following chapters will get into the details of how to evaluate this work –Chapter 7– and the results obtained with it –Chapter 8.

6.4 Discussion

This chapter has presented the changes needed within both PREESM and Apollo to fully integrate both domains, thus completing the design of the methodology envisioned in this work. As stated in the introduction of this chapter and depicted in Figure 6-1, the new functionalities added to address this integration can be grouped into three domains, and their main contributions can be summarized as follows:

- **Polyhedral-dataflow interface:** An API to set up the communication channels between polyhedral and dataflow domains has been created, together with the necessary mechanisms to make Apollo reactive to dataflow instructions.
- **Dataflow preparation:** This category includes the changes added within Apollo to make it dataflow-compliant. These changes can be summarized as follows: (i) creation of different working modes within Apollo to limit its multithreaded activity; (ii) adaptation of Apollo and all the underlying tools to use them in multithreaded environments; and (iii) creation of a unique identifier to make Apollo able to distinguish any kernel coming from the dataflow application.
- **Polyhedral preparation:** Finally, so as to make a dataflow application use Apollo, some changes need to be addressed beforehand:
 1. **Framework-level changes:** For Apollo to work properly, the compiling toolchain of both PREESM and SPiDER has been modified.
 2. **Application-level changes:** The usage of the API has been automated within PREESM and SPiDER so the user only needs to activate Apollo.
 3. **Actor-level changes:** The passing of the configuration of each actor from the dataflow domain to the polyhedral one has also been automated within PREESM and SPiDER and made dependent on the activation of Apollo. Once this is done, the last step is including Apollo's dedicated pragma surrounding the targeted kernels. This step must be performed manually by users, so a tutorial is available in PREESM website to guide them.

These changes complete the design and implementation of the methodology proposed in this document. The rest of the chapters of this document are devoted to envision a heterogeneous set of applications with which this methodology can be validated: first, Chapter 7 presents this set of applications; afterwards, Chapter 8 describes the experiments carried out and discusses the obtained results.

7

VALIDATION

As described in Section 3.2, the new methodology proposed in this document to ease the acceleration of dataflow-based applications is going to be validated following a two-step process: first, the methodology is going to be functionally validated through a set of benchmarks widely used in the domains involved in this document –i.e., dataflow-based design and automatic parallelization techniques–; afterwards, this methodology will be validated through a real-life application to show that the advantages hypothetically thought for the methodology are actually present when using this methodology to model a real-life application.

To that end, Section 2.3 and Section 2.4 reviewed the state of the art of the different benchmarks and the field the real-life application selected belongs to, respectively; likewise, Section 3.2 justified the choices made to this respect, composing the final set of benchmarks and applications that will be used to validate this methodology.

This chapter will continue this line by describing in detail the different kernels conforming each of the benchmark suites selected –Section 7.1– and introducing the real-life application that has been selected as use-case –Section 7.2–, providing a description of the algorithms involved.

7.1 Functional validation: benchmarks

As stated in Section 3.2, the functional validation of the proposed methodology will be carried out using a set of benchmark suites coming from the different domains in which this methodology is embedded: dataflow-based design and automatic parallelization techniques.

- From the dataflow-based design domain, the applications specifically designed by PREESM developer community to test new functionalities will be used to validate the solution proposed in this document from the dataflow point of view.
- From the automatic parallelization domain, two different benchmark suites are going to be used: **PolyBench** and **ApolloBench**. The former is the benchmark suite most widely used in automatic parallelization and, more specifically, in the polyhedral domain, while the latter is the benchmark suite used originally by Apollo to validate the tool, since it includes features not covered by **PolyBench**, such as pointers, indirections or unpredictable loop bounds.

The remaining of this section is going to get into the details of these benchmark suites, providing a brief description of each kernel within them. Subsection 7.1.1 will provide an overview of the applications that will be used as benchmarks for the dataflow domain, while Subsection 7.1.2 will depict the benchmark suites selected from the automatic parallelization domain.

7.1.1 Dataflow-based benchmarks

As already established, to functionally validate the proposed methodology from the dataflow point of view, some of the most typically used applications by PREESM developers to validate new functionalities are used: *Sobel-morpho* [Georgakarakos'17], *Stereo matching* [Pelcat'14, Madronal'19] and *Matrix multiplication* [Giorgi'16].

- **Sobel-morpho:** Sobel operator, also known as Sobel-Feldman operator, is a filter widely used in image processing and computer vision domains, and, more specifically, as part of edge detection algorithms, generating images in which the edges are emphasized. From the technical side, it is a discrete differentiation operator that estimates the gradient of the intensity function of the image so as to provide either the associated gradient vector or its norm for every point in the image [Sobel'68]. Since its main application field is image processing, it makes it a suitable candidate for dataflow specifications, and it has been extensively used by PREESM developers for evaluation purposes. In this case, the application is coupled with *erosion* and *dilation* morphological operations.
- **Stereo matching:** The goal of the stereo matching –or disparity estimation– algorithm is to process a couple of images captured by two different cameras to generate a disparity map expressing the depth of the scene, i.e., the third dimension [Pelcat'14]. It is a common resource in computer or robotic vision to determine three-dimensional depth information, which in turn makes it a suitable algorithm for autonomous vehicles, virtual reality or robot navigation [Mondal'17]. As also stated in [Pelcat'14], two main features turn this application into a very interesting case study to evaluate PREESM rapid prototyping framework: the large computational load and memory requirements of this kind of algorithm, and its potential use in Advanced Driver Assistance Systems (ADAS)¹. More details about the stereo matching algorithm and, more precisely, of its implementation in PREESM, can be found in [Pelcat'14].
- **Matrix multiplication:** Matrix multiplication is one of the most used applications to characterize automatic parallelizers. It belongs to the dense linear algebra category, which has been traditionally used to assess the efficiency and performance of new architectures [Volkov'08, Heinecke'13, Giorgi'16]. Within this category, matrix multiplication is selected because it is one of the most fundamental tasks in computer science and mathematics, and many examples of its usage can be found in the literature [Le Gall'14, Salim'15, Hemeida'20].

¹<http://www.ti.com/lit/wp/spry251/spry251.pdf>

7.1.2 Automatic parallelization benchmarks

As stated at the beginning of this section, two different benchmark suites, **PolyBench** and **ApolloBench**, have been chosen to be used to functionally validate the methodology proposed in this document. This section is going to review both of them, providing a brief description of each of the kernels composing them and highlighting the challenges they pose to the framework.

PolyBench

As established in its website², PolyBench is a set of kernels including SCoPs, a concept previously described in Subsection 2.2.4 and which can be defined as a part of a program that is compliant with the polyhedral model. This benchmark suite contains 30 numerical computations coming from diverse application domains, such as linear algebra computations, statistics, image processing, dynamic programming or physics simulation, and are currently provided both in C and Fortran.

Table 7-1 provides a summary of the kernels included in PolyBench, sorted by the domain to which each kernel belongs, and together with a brief description of each one. This information has been extracted from the official documentation of PolyBench, which is open-access and available online³.

As mentioned in Section 3.2, PolyBench has been widely used in the polyhedral model domain, being the reference benchmark suite for many compiler research works. To those works, PolyBench offers advantageous features; for instance, it offers the possibility to exploit the usage of the different levels of memory by establishing 5 predefined problem sizes for the input data of each of the benchmarks, gathered in Table 7-2. They are specifically designed to make the input data fit in one of the different caches: L1, L2 and L3 –MINI, SMALL and MEDIUM– or, conversely, to force it to exceed the size of the caches –LARGE and EXTRALARGE.

ApolloBench

As established in Section 3.2, ApolloBench is the customized benchmark suite used to originally evaluate Apollo. It was built as a set of different kernels, extracted in turn from other benchmark suites of the literature. These kernels were selected to highlight the different capabilities of Apollo, so they can contain indirections, pointers, unpredictable bounds or unpredictable scalars. The main characteristic defining these kernels is that they are not compliant with the polyhedral model at compile time, so they cannot be targeted by traditional automatic parallelization frameworks, such as those based on the polyhedral model. However, they can be compliant with this model at runtime, which was the objective for which Apollo was originally developed.

As described in [Caamaño'16], ApolloBench is composed of 8 kernels: `mri-q` from the **Parboil** suite [Stratton'12], `needle` and `backprop` from the **Rodinia** suite

²<https://sourceforge.net/p/polybench/wiki/Home/>

³<https://sourceforge.net/projects/polybench/files/polybench-c-4.2.1-beta.tar.gz/download>

Table 7-1: PolyBench kernels. Extracted from PolyBench official documentation.

Domain	Benchmark	Description
Data mining	correlation	Correlation computation
	covariance	Covariance computation
BLAS routines	gemm	Matrix multiplication: $C = \alpha \times A \times B + \beta \times C$
	gemver	Vector multiplication and matrix addition
	gesummv	Scalar, vector and matrix multiplication
	symm	Symmetric matrix multiplication
	syr2k	Symmetric rank-2k operations
	syrk	Symmetric rank-k operations
	trmm	Triangular matrix multiplication
Linear algebra kernels	2mm	2 matrix multiplications ($\alpha \times A \times B \times C + \beta \times D$)
	3mm	3 matrix multiplications: $((A \times B) \times (C \times D))$
	atax	Matrix transpose and vector multiplication
	bicg	BiCG Sub kernel of BiCGStab linear solver
	doitgen	Multi-resolution analysis kernel
	mvt	Matrix-Vector product and transpose
Linear algebra solvers	cholesky	Cholesky decomposition
	durbin	Toeplitz system solver
	gramschmidt	Gram-Schmidt decomposition
	lu	LU decomposition
	ludcmp	LU decomposition followed by forward substitution
	trisolv	Triangular solver
Medley	deriche	Edge detection filter
	floyd-warshall	Shortest path detector
	nussinov	Dynamic programming for sequence alignment
Stencils	adi	Alternating Direction Implicit solver
	fdtd-2d	2-D Finite Different Time Domain kernel
	heat-3d	Heat equation over 3D data domain
	jacobi-1d	1-D Jacobi stencil computation
	jacobi-2d	2-D Jacobi stencil computation
	seidel	2-D Seidel stencil computation

Table 7-2: Summary of the different sizes predefined by **PolyBench** for each kernel. Extracted from PolyBench official documentation.

Label	Size	Memory hierarchy
MINI	< 16 KB	L1 cache
SMALL	$\approx$ 128 KB	L2 cache
MEDIUM	$\approx$ 1 MB	L3 cache
LARGE	$\approx$ 25 MB	-
EXTRALARGE	$\approx$ 120 MB	-

[Che'09], `sor` from the **Scimark** suite⁴, and `pcg`, `ispmatmat`, `dmatmat` and `spmatmat` from **SPARK00** suite [Spek'08]. Table 7-3 provides a description of each kernel, while Table 7-4 completes this information by highlighting their most interesting features from Apollo's point of view. Five different inputs are used for `spmatmat` to represent different data layouts for the sparse matrix: a diagonal matrix, a square matrix, a matrix with memory accesses modeled as *tubes*, and two matrices providing cases that cannot be optimized by Apollo.

Table 7-3: ApolloBench kernels. Extracted from [Caamaño'16].

Benchmark	Description
<code>mri-q</code>	Computation of a matrix Q used in 3D MRI reconstruction algorithms
<code>needle</code>	Needleman-Wunsch optimization method for DNA sequence alignments
<code>sor</code>	Jacobi's Successive Over-Relaxation with grid averaging memory patterns
<code>backprop</code>	Neural Network training algorithm with unstructured grids
<code>pcg</code>	Preconditioned Conjugate Gradient, iteratively solving $Ax = b$
<code>dmatmat</code>	Dense matrix times dense matrix
<code>ispmatmat</code>	Sparse matrix times dense matrix
<code>spmatmat-diag</code>	Sparse matrix (diagonal) times dense matrix
<code>spmatmat-rdm</code>	Sparse matrix (random) times dense matrix
<code>spmatmat-square</code>	Sparse matrix (square) times dense matrix
<code>spmatmat-tube</code>	Sparse matrix (tube) times dense matrix
<code>spmatmat-worst</code>	Sparse matrix (worst) times dense matrix

Table 7-4: Characteristics of ApolloBench kernels. Extracted from [Caamaño'16].

Benchmark	Has indirections	Has pointers	Unpredictable bounds	Unpredictable scalars
<code>mri-q</code>		✓		
<code>needle</code>		✓		
<code>sor</code>	✓	✓		
<code>backprop</code>	✓	✓		
<code>pcg</code>	✓	✓	✓	✓
<code>dmatmat</code>	✓	✓		
<code>ispmatmat</code>	✓	✓	✓	✓
<code>spmatmat</code>	✓	✓	✓	✓

Finally, as happened with PolyBench, ApolloBench kernels are also configured to operate with different problem sizes, being the predefined ones SMALL, MEDIUM and LARGE. ApolloBench can be downloaded from Apollo's official website⁵.

⁴<https://math.nist.gov/scimark2/>

⁵<https://gforge.inria.fr/frs/download.php/latestfile/5209/apollo-benchs-1.0.0.tar.gz>

7.2 Real use-case validation: application

As established at the beginning of this chapter, its objective is to provide the necessary information about the applications used to validate the methodology proposed so as to fully understand Chapter 8, which analyzes the obtained results.

To that end, the rest of the section provides the necessary information to get into the details of the selected real-life use-case application, whose objective is to help surgeons in delimiting brain tumor boundaries during surgical procedures.

To do so, first the context from which this application was born, HELICoiD project, is going to be introduced –Subsection 7.2.1–; afterwards, the database that is going to be used will be presented –Subsection 7.2.2–; finally, the algorithms composing the application will be outlined –Subsection 7.2.3.

7.2.1 HELICoiD project

As already introduced in Subsection 2.4.2, HELICoiD was an interdisciplinary, cutting-edge project funded by the Research Executive Agency (REA) through the Future and Emerging Technologies (FET)-Open call, under the 7th Framework Programme (FP7) of the European Union [Salvador’17]. The project was developed as a collaboration among two hospitals, three industrial partners and four universities, and its goal was to devise and build a HSI demonstrator to discern between healthy and tumor brain tissue during neurosurgical procedures in real-time. Such a demonstrator can provide surgeons with crucial information to accurately determine tumor boundaries, assisting them in (i) avoiding the excessive extraction of healthy tissue, and (ii) preventing tumor remnants from being left behind. Accurately locating tumor limits is expected to affect, in turn, the outcome of the surgery and the expected prognosis of the patient [Fabelo’18a]. Additionally, this demonstrator constitutes a non-contact, non-invasive and non-ionizing approach, which is an added value for surgeons.

Figure 7-1 shows a block diagram of the demonstrator developed within this project –Figure 7-1a– together with an image captured at the University Hospital Doctor Negrin of Las Palmas de Gran Canaria (Spain), showing the demonstrator being used during a surgical procedure. The customized HSI acquisition system is composed by:

- A Visible-to-Near-Infrared (VNIR) hyperspectral camera using a pushbroom mechanism to scan 826 spectral bands in the range between 400 and 1000 nm.
- An illumination device generating cold light in the range between 400 to 2200 nm thanks to a Quartz Tungsten Halogen (QTH) lamp with the goal of avoiding high temperatures upon the exposed brain surface.
- A high-precision stepper motor that controls the pushbroom scanning.
- A processing unit in charge of controlling all the components of the system.

Figure 7-1: HELICoiD demonstrator: (a) block diagram and (b) example of its usage during a neurosurgical procedure at the University Hospital Doctor Negrin of Las Palmas de Gran Canaria, Spain. Extracted from [Florimbi'18].

7.2.2 HELICoiD database

Apart from the design and verification of the demonstrator, another relevant outcome of HELICoiD was the creation of the first open-access HSI database specifically developed for in-vivo human brain tumor delineation [Fabelo'19b]. The data acquisition for building this database was carried out in two different campaigns, from March 2015 to June 2016, at the University Hospital of Southampton (United Kingdom) and at the University Hospital Doctor Negrin of Las Palmas de Gran Canaria (Spain). The patients selected for this study were all adults –18 years old and above– and undergoing craniotomy for resection of intra-axial brain tumors, either primary or secondary. Additionally, patients undergoing resection of non-superficial brain tumors were also included in the study, so as to properly capture the exposed healthy brain tissue. To participate in this study, all the subjects gave written informed consent.

As aforementioned, this database is open-source, so it is available online⁶. It contains 36 in-vivo brain surface images captured from 22 different patients, together with a golden reference conformed by some pixels labeled as tumor tissue, normal tissue, blood vessels and other tissues, materials or substances not relevant for the surgical procedure, but that can be present in the image –hereafter *background*–. As explained in [Fabelo'19b], nowadays it is not possible to provide a complete gold reference map of the captured scene with absolute certainty that the pixel represents the associated class. To do that, the entire brain tissue exposed should be assessed by a pathologist, which in turn means that the surgeon should resect all the tissue exposed in the brain surface, thus compromising the patient's health to a great extent. For those reasons, this database also provides a methodology to achieve a partial gold standard

⁶<https://hsibraindatabase.iuma.ulpgc.es/>

map of its images based on visual inspection of the scene by specialists together with a robust labeling tool, which in this case is based on the Spectral Angle Mapper (SAM) algorithm [Kruse'93], so as to provide a highly reliable four-class labeled dataset.

7.2.3 HELICoiD processing algorithm

As described in Subsection 7.2.1, the hyperspectral camera integrated in HELICoiD demonstrator captures information in 826 spectral bands. As HELICoiD objective is to provide surgeons with useful information about the tumor boundaries, the images captured by the demonstrator must be processed in real-time –or in *intraoperative real-time*, which is defined as the time the camera takes to capture an image.

For that reason, a customized classification framework was developed within the context of HELICoiD project to process the hyperspectral images obtained during the procedure and provide surgeons with the desired results [Fabelo'18b]. The goal of this framework is to take advantage of both the spatial and spectral features of hyperspectral images through a hybrid classification framework that combines both supervised and unsupervised classification techniques. Figure 7-2 presents the block diagram of this classification framework, while Figure 7-3 presents the expected outputs at certain points of each stage of the hybrid classification. As can be observed, this hybrid classification framework consists of four main stages: preprocessing, supervised classification, unsupervised classification and Majority Voting (MV).

Figure 7-2: Block diagram of HELICoiD processing chain. Based on a figure from [Salvador'17].

- **Preprocessing:** Once the hyperspectral image has been captured, the raw HSI cube needs to be preprocessed so as to homogenize the spectral signatures of each pixel. To do so, the preprocessing routine developed in the context of this project is composed of five steps: calibration, noise filtering, extreme noise band removal, band averaging and normalization [Fabelo'16a]. After this process, a reduced HSI cube is obtained, with only 128 bands.
- **Supervised classification:** After obtaining the preprocessed cube, the labeling tool is used by the specialists to obtain the golden standard needed to build the supervised classification model. Once the model is extracted, it is used to perform the supervised classification of the preprocessed cube. As

Figure 7-3: HELICoiD results: representation of the input, the partial results and the final classification provided to the surgeon.

shown in Figure 7-2, this classification is divided, in turn, into three main steps: dimensionality reduction by means of Principal Component Analysis (PCA) algorithm, pixel-wise classification by means of SVM algorithm, and classification refinement by means of K-Nearest Neighbors (KNN) filtering algorithm. A possible output of this stage is observed in Figure 7-3b.

- **Unsupervised classification:** This classification, also known as unsupervised clustering segmentation, aims at exploring different subspaces within the HSI cube so as to provide a segmentation map where all the different materials found in the image are grouped into several clusters with similar spectral features, together with their correspondent cluster *centroid*. Each centroid represents a spectrum corresponding to a material in the scene, but, since it is unsupervised, it cannot identify the nature of the material. In the context of this project, the unsupervised classification is carried out by means of Hierarchical K-Means (HKM) algorithm, since it provides a good delimitation of the different areas appearing in the scene that should be identified either by a specialist or by an automatic process, such as the supervised classification. An example of a possible output of this stage is shown in Figure 7-3c.
- **Majority Voting (MV):** As mentioned before, unsupervised clustering can provide some relevant information to an automatic process such as the supervised classification described above, so a method to merge both results is required to obtain the final hybrid classification map. This combination is performed by means of the MV algorithm, since it has been previously used with successful results in HSI [Tarabalka'09]. This algorithm takes as input both the supervised and the unsupervised classification results, merging them as follows: during the clustering algorithm, all the pixels belonging to each found cluster are assigned to the same class; then, to select the class, the supervised classification map is analyzed and the most frequent class in each cluster is selected. An example of the expected output is shown in Figure 7-3d.

As can be seen, the approaches involved in this classification are computationally complex; hence, one of the technical challenges of this project is processing the huge volume of data the camera captures in intraoperative real-time. Only by reaching real-time performance does the system become useful to the surgeon, so, in turn, the usage of parallelization techniques becomes compulsory [Salvador'17].

As established in [Fabelo'18b], due to the high computational load of the hybrid classification framework, HELICoiD consortium decided to offload some parts of the framework to a hardware accelerator, namely the supervised classification, since it is the one demanding the most computational resources and, hence, the one requiring the highest level of parallelism exploitation.

For those reasons, the supervised classification approach has been chosen as the application that is going to be used during this document as a real use-case to validate the methodology presented in this work. The rest of this section is going to introduce the three algorithms involved in this supervised classification approach. It should be noted that this information has been extracted from [Lazcano'19b].

As aforementioned, the supervised classification stage of HELICoiD is mainly composed by a pixel-wise classifier, implemented through SVM algorithm, which is afterwards refined and smoothed through a filtering step, performed by KNN algorithm. However, as KNN takes as input a one-band representation of the image together with the classification map provided by SVM, a third step, responsible for generating this one-band representation, must be added. Hence, the supervised classification stage is completed with a dimensionality reduction algorithm, such as PCA. The block diagram associated to this stage is shown in Figure 7-4, together with the expected output of each algorithm.

Figure 7-4: Block diagram of HELICoiD supervised classification stage. Extracted from [Lazcano'19b].

Support Vector Machine (SVM)

As already stated in Subsection 2.4.3, the goal of pixel-wise classifiers, such as SVM, is to categorize the contents of each pixel of a captured scene through the information contained in its spectrum [Camps-Valls'05]. The main characteristic of pixel-wise classifiers is that only the pixel spectrum is considered, so each pixel can be classified independently. Apart from just associating a class to each pixel of the image, SVM can go further and assess the probability of each pixel to belong to the different elements –or classes– appearing in the scene. As a result, SVM can be decomposed in two steps: training and classification. Although these steps are going to be summarized below, more details can be found in [Madroñal'17b].

- **Training:** The objective of this step is twofold: first, to study the gold reference map of the image, which basically is a set of samples that have previously been labeled by a specialist–e.g., a pathologist–; then, to generate a set of hyperplanes to discern between each pair of classes under analysis, applying a one-vs-one strategy. These hyperplanes –also known as *binary hyperplanes*– are computed by maximizing the distance between the two classes involved [Melgani'04].
- **Classification:** The goal of this step is also twofold: to define the class to which each pixel of the image belongs, and to compute the probability of each pixel to belong not only to the chosen class, but to all the rest. This process can be summarized in three phases, all of them following a pixel-wise procedure: first, the distances from each pixel to each binary hyperplane are computed, together with their associated binary probabilities; after that, those binary probabilities are combined as described in [Wu'04] to minimize the complexity of the implementation; finally, the class with the highest probability is selected as that to which the pixel belongs. With this information, a first classification map is generated. However, as this algorithm is going to be used coupled with a spatial filtering process, the data that is going to be fed to the filtering algorithm –i.e., KNN– are the probabilities computed for each pixel.

Principal Component Analysis (PCA)

As already mentioned in this document, one of the main challenges of HSI is that it generates huge volumes of information that most times must be processed under tight performance constraints. However, adjacent bands usually contain deeply correlated information, which can be used to reduce the amount of information contained within an image. To that end, PCA aims at minimizing this correlation by eliminating the redundancies between adjacent wavelengths. To do that, this algorithm applies a transformation based on the eigenvectors of the covariance matrix extracted from the original input image. Since, as established above, the goal of using PCA in the context of HELICoiD is to get a one-band representation of the original image, only the first component is kept once the original image has been projected onto the eigenvectors. As happened with SVM, although a brief summary of PCA is provided, an extended description can be found in [Lazcano'17c].

Consequently, PCA can be divided into four sequential steps: first, the average of each band is computed and removed to obtain the so-called centered image; secondly, the covariance matrix associated to this centered image is obtained; once the covariance matrix has been obtained, its associated eigenvectors are extracted; finally, the original image is projected onto the eigenvectors subspace and the first component is retained. Within the context of HELICoiD project, the method selected to compute the eigenvectors is the Jacobi approach, which is an iterative process that applies planar rotations to the greatest off-diagonal element to make it zero, successively approximating the input matrix to a diagonal one containing the eigenvalues. This iterative process is stopped once all the off-diagonal elements are below a certain threshold or stop condition, which is defined a priori. A full detailed analysis of Jacobi method is provided in [Lazcano'17c].

K-Nearest Neighbors (KNN)

As shown in the recent literature, KNN filtering algorithm has been widely used in HSI machine learning applications [Ma'14], and there are several examples applying this filter to improve the classification results obtained with supervised classifiers as SVM by exploiting both spatial and spectral features of hyperspectral images [Huang'16, Fabelo'18b]. To that end, KNN refines the probability maps obtained with SVM algorithm by matching and averaging non-local neighborhoods. In a nutshell, for each pixel in the image, KNN: (i) computes the K nearest neighbors, being K a constant defined a priori; and (ii) averages the probabilities estimated by SVM. As happened with PCA and SVM, the remainder of this section is going to provide an overview of the algorithm, but a full analysis can be found in [Florimbi'18].

To find the K nearest neighbors of a certain pixel, KNN first needs to compute the distance from that particular pixel to every other in the image. To do that, the algorithm requires the spatial coordinates of the pixels in the image together with the spectral value of that particular pixel, which is extracted from the one-band representation of the image generated by PCA. A balance parameter, λ , weighs these two pieces of information: if $\lambda = 0$, the algorithm is oblivious to the spatial information; conversely, the higher the value of λ , the higher the influence the local neighborhood presents. With respect to the number of neighbors to be used, K , increasing its value increases the computational cost, since more neighbors need to be estimated. For these reasons, in the context of HELICoiD project these parameters have been fixed to $\lambda=1$ and $K=40$; previous studies have experimentally proven these values to be a good compromise, as larger values may over-smooth the result [Fabelo'18b]. It should be noted that the Euclidean distance –or *squared 2-norm*– is used to estimate distances. Equation 7-1 shows how this method computes the distance from a certain pixel with coordinates (r, c) to any other pixel with coordinates (i, j) . Therefore, being I the one-band representation of the image, I_{rc} represents the normalized pixel value of I for row r and column c , while I_{ij} represents the same value, but for row i and column j .

$$d(I(rc), I(ij)) = (I_{rc} - I_{ij})^2 + (r - i)^2 + (r - j)^2 \quad (7-1)$$

When the K nearest neighbors are all computed, the filtered map, O , is generated through the evaluation of the expression gathered in Equation 7-2. In this expression, P stands for the original probability map generated by SVM algorithm, while w_i denotes the K nearest neighbors for pixel i .

$$O(i) = \frac{\sum P(j)}{K}, j \in w_i \quad (7-2)$$

Consequently, the total number of O output probability maps will be equal to the number of classes previously defined in SVM algorithm. These optimized probability maps are in turn used to generate a final classification map, which is obtained by assigning the label of the class with the highest probability to each pixel in the image and represents the output of the supervised classification stage of HELICoiD.

8

RESULTS

Up until now, the rest of chapters have introduced the methodology proposed, the state of the art related to it, the details of how it has been finally developed and combined with other existing methodologies and, finally, how to validate it so as to both prove its correctness and highlight the many advantages it can offer. To close this document, this chapter will thoroughly present, analyze and discuss the different experiments that have been carried out in order to perform this double validation.

This chapter is divided into four sections: first, Section 8.1 describes the setup devised for the experiments, outlining the platforms used and the transformations evaluated in the training phase of the multiversioning system; afterwards, Section 8.2 presents the experiments carried out to address the functional validation of the methodology with the benchmarks presented in Chapter 7; likewise, Section 8.3 does the same with the experiments addressed for the use-case validation; finally, Section 8.4 closes the chapter summarizing the most relevant results.

8.1 Setup

This section presents the context in which the experiments have been carried out. To do so, three main points are reviewed: (i) the architectures upon which the different experiments have been run; (ii) the list of different optimizing transformations to be evaluated by the multiversioning system during its training phase, as presented in Chapter 5; and (iii) the configuration of the different Apollo working modes depending on the tests being carried out, which were introduced in Section 6.2.

With respect to the target platforms, the experiments presented during this chapter have been run onto two different hardware architectures:

- **Namaka:** A general-purpose multicore architecture that gathers an AMD Ryzen™ Threadripper™ 1950X processor with 16 cores running at 3.4 GHz and with Linux 4.15.0-65. The RAM memory size is 32 GB, while the cache sizes are 32KB of L1d, 64KB of L1i, 512KB of L2 and 8192KB of L3.
- **Magerit**¹: A cluster composed of 68 Lenovo ThinkSystem SD530 nodes, each one gathering two Intel Xeon Gold 6230 processors with 20 cores running at 2.1 GHz

¹The candidate gratefully acknowledges Universidad Politécnica de Madrid (www.upm.es) for providing computing resources on Magerit Supercomputer.

and with Linux 3.10.0-957. The RAM memory size is 192 GB, while the cache sizes are 32KB of L1d, 32KB of L1i, 1024KB of L2 and 28160KB of L3. The experiments have been conducted on the 20 cores of one Intel Xeon Gold 6230 processor, with one thread per core, using 4 GB of RAM per core.

In general lines, it should be noted that all the experiments that are going to be presented in this section have been conducted disabling thread migration, so only one thread can be run per core. This decision has been taken so as to correctly illustrate the goodness of the multiversioning system, since otherwise the numbers provided may not be representative. On the one hand, with respect to Namaka, the experiments are going to use the 16 cores of the processor; on the other hand, the experiments executed in Magerit are going to use only the 20 cores of one processor embedded in one of the 68 clusters. The rationale behind this decision is that, by using only one processor, communications are avoided, and the comparison with Namaka can be fairer.

With respect to the transformations that are going to be tested, Table 5-1 presented the different flags that can be used to guide Pluto in the generation of the optimizing transformation to test –repeated here in Table 8-1 to ease the readability of the document–. As can be observed, some flags are devoted to tiling tuning, while others are devoted to playing with unrolling or to consider also RAR dependences. Two special flags have to be highlighted: on the one hand, `-parallel` does not configure any transformation, but allow Pluto to insert OpenMP directives in the code to parallelize loops; on the other hand, `-identity` just checks if the loop can be directly parallelized and, if so, adds the corresponding OpenMP directives, but without applying any further transformation. This flag is interesting because it minimizes Pluto’s overhead, since the tasks it must perform are very limited. Hence, it is a good candidate when the developer wants to generate a transformation as fast as possible.

Table 8-1: List of Pluto flags (repeated from page 72).

Flag	Description
<code>--tile</code>	Activate tiling
<code>--intratileopt</code>	Optimize intra-tile execution, allowing transformations as loop interchange
<code>--l2tile</code>	Apply tiling a second time (usually for optimizing L2 cache usage)
<code>--diamond-tile</code>	Apply load-balanced tiling, specifically thought for stencil computations
<code>--parallel</code>	Parallelize automatically (insert OpenMP directives)
<code>--rar</code>	Consider RAR dependences, not only WAR and RAW
<code>--unroll</code>	Activate loop unrolling
<code>--ufactor</code>	Factor controlling loop unrolling
<code>--prevector</code>	Enable transformations to make code amenable to vectorization
<code>--identity</code>	Parallelize the loop kernel by slicing each outermost loop into parallel threads, without applying any further transformation

Besides these flags, Pluto also considers the possibility of varying the tiling size, although it cannot be controlled by a flag. For that reason, a mechanism to change the tiling size has been added to Apollo with the objective of evaluating different tiling

sizes to find the most adequate one, depending on the kernel that is being evaluated and the architecture upon which the kernel is being executed.

As shown in Table 8-1, Pluto provides many possibilities for tuning the transformation; however, in order for the multiversioning system to be useful, the training phase must be limited to a few transformations. For that reason, the multiversioning system has been set up to test 15 transformations whenever the training phase is launched. Table 8-2 presents the different combinations that are going to be tested in the experiments presented in this chapter.

As established in Chapter 5, the set of transformations to test is easily modifiable, so users can refine the transformations as they see fit. The combinations chosen in this document, and depicted in Table 8-2, have been selected to explore the transformation space and prove the advantages of a multiversioning system, trying to build a diverse set that sufficiently represents all the possible transformations, while keeping the number of transformations to test within reasonable limits. Yet, it should be noted that this set of transformations has been chosen heuristically; since a thorough study and analysis of the transformation space is out of scope of this work, it is proposed as a future research line, as mentioned in Section 9.4.

Table 8-2: Pluto flag combinations used for multiversioning.

ID	Flag combination	Description
0	no transformation	Original code
1	identity	Fast Pluto: identity transformation
2	intratile	Intratile to allow loop interchange
3	tile $32 \times 32 \times 32$ + intratile	Default tiling
4	tile $32 \times 32 \times 32$ + intratile + unroll $\times 2$	Default tiling and unroll by 2
5	tile $32 \times 32 \times 32$ + intratile + unroll $\times 8$	Default tiling and unroll by 8
6	tile $16 \times 16 \times 16$ + intratile	Tiling (tile size 16)
7	tile $64 \times 64 \times 64$ + intratile	Tiling (tile size 64)
8	tile $128 \times 128 \times 128$ + intratile	Tiling (tile size 128)
9	tile $256 \times 256 \times 256$ + intratile	Tiling (tile size 256)
10	tile $32 \times 32 \times 64$ + intratile	Heterogeneous tiling
11	tile $64 \times 64 \times 32$ + intratile	Heterogeneous tiling
12	tile $16 \times 64 \times 32$ + intratile	Heterogeneous tiling
13	diamond-tile + intratile	Diamond tiling (stencil computations)
14	tile $32 \times 32 \times 32$ + intratile + l2tile	Default tiling and L2 tiling

As can be seen, it has been decided that the original code will be also considered as a candidate, since there can be cases in which the original code remains the best option, namely with very low workloads. The rest of the transformations are devoted to test different features that may have a considerable impact in the performance.

- **Tiling size:** Combinations 3 to 12 are devoted to playing with different tiling sizes. The first one to be tested is 32, since it is the size defined by default by Pluto. Regarding the terminology used, a tiling of $32 \times 32 \times 32$ means that, supposing

a kernel containing three nested loops, a tiling of 32 is applied within each level. As for the heterogeneous tiling, this rule still applies but, should more than three nested loops appear, from the third on the tiling size applied is the last one.

- **Identity:** As already mentioned, Pluto presents a flag that can be used to just check whether the kernel under study can be parallelized or not and, if so, add the corresponding OpenMP directives. This operation can be performed really fast by Pluto, since it does not involve any kind of transformation. For that reason, combination 1 has been added, since it can provide some good results while minimizing the overhead added by Pluto.
- **Loop interchange:** As described in Table 8-1, `intratile` flag is useful to enable transformations such as loop interchange. Hence, this flag is always activated; yet, combination 2 specifically tries this flag without any tiling, since there can be cases in which tiling is not useful, but a loop interchange can still be applied.
- **Unrolling:** Combinations 4 and 5 test unrolling transformations with different unrolling factors, namely 2 and 8, being the former Pluto's default value.
- **Diamond tiling:** Pluto counts with a specific flag to enable load-balanced tiling –i.e., diamond tiling–, which is a type of tiling particularly useful for stencil codes. As PolyBench contains a specific section with several stencil kernels, this combination has been chosen to evaluate its effect on the performance.
- **L2 cache optimization:** Combination 14 is devoted to trying to optimize the L2 cache usage. Although it has been reported that this flag can considerably slow down certain executions [Caamaño'16], for others it may provide a significant speedup; consequently, it has been chosen as a possible transformation to try to discern a pattern of codes upon which this flag could be useful.

As for the rest of the flags appearing in Table 8-1 that have not been mentioned, it can be due to three main reasons: (i) the flags have a default configuration –as `prevector`, which is aligned with tiling and, thus, inherits its behavior–; (ii) they are configured externally, and the same configuration is going to hold for all the transformations –as is the case of `parallel`, which can be activated or deactivated from the dataflow system depending on which framework is going to hold the parallelization capabilities; or (iii) the flags have not been considered in the final combination set because the advantages they can provide are somehow not as good as other flags –as, for instance, flag `rar`, which can slightly improve data locality, but many other flags already do that–. This selection has been made so as to build a diverse set, but while keeping its size within reasonable limits; since the system is thought to be used at runtime, the larger the training set, the larger the overhead the operational phase must cancel out, thus worsening the efficiency of the system.

Finally, to conclude this section, the different modes presented in Section 6.2 need to be configured for each experiment. As summarized in Table 6-2, 5 different modes of functioning have been envisioned: (i) normal behavior of Apollo; (ii) timeout of

X times the time of the sequential version; (iii) absolute timeout of Y seconds; (iv) absolute timeout of Y seconds, but without launching the sequential thread; and (v) safeguard to execute only the sequential version. These modes are going to be tested throughout this chapter as gathered in Table 8-3:

- Mode 1 denotes Apollo original behavior, so it does not need to be validated.
- Modes 2 and 3 are validated in Subsection 8.2.1, as they are used to evaluate the multiversioning system so as to show how the most suitable transformation, as well as the achieved performance, vary when different timeouts are tested.
- Modes 4 and 5 will be validated both in Section 8.2 and Section 8.3, since they have been specifically developed so as to build a dataflow-oriented Apollo and, thus, they will be used to validate the methodology both from the dataflow point of view and with a real-life application.

Table 8-3: Summary of the different modes defined in Apollo.

Mode	Sequential thread	Timeout	Function	Validation
Mode 1	✓	-	Apollo original behavior	No validation required
Mode 2	✓	$N \times$	$N \times$ timeout	Subsection 8.2.1
Mode 3	✓	Y seconds	Y seconds timeout	Subsection 8.2.1
Mode 4	✗	Y seconds	PREESM specific usage	Section 8.2 and Section 8.3
Mode 5	✗	-	Safeguard mechanism	Section 8.2 and Section 8.3

8.2 Functional validation results

In this section, the evaluation of the proposed methodology through a set of benchmark suites is going to be carried out. This evaluation is going to be divided into two main groups: first, Subsection 8.2.1 will evaluate the methodology from the polyhedral domain point of view with the objective of validating the new functionalities added to Apollo, namely the multiversioning system; afterwards, Subsection 8.2.2 will evaluate the methodology from the dataflow domain perspective.

8.2.1 Polyhedral benchmarks

As introduced in Chapter 7, two different benchmark suites are used to validate the extensions added to Apollo: **PolyBench**, which is the benchmark suite most widely used in the polyhedral domain, and **ApolloBench**, which was specifically composed to originally evaluate Apollo. To sum up, this section presents the following ideas:

- **Specific setup:** Before starting the study, the benchmarks are going to be refined. Since extending Apollo's polyhedral model support at runtime is out of the scope of this PhD, those benchmarks for which Apollo fails to apply this model are going to be removed (if any). If so, the reasons why those tests fail are going to be stated, discerning between cases in which Apollo does not yield a result, but the multiversioning system does; and cases in which neither Apollo nor the multiversioning system can be expected to yield a satisfactory result. As a result, the latter will not provide any useful information for the study carried out in this chapter, so they are going to be removed from the final benchmarks set. This assessment will be addressed in Subsubsection 8.2.1.1.
- **Transformation selection:** Once the final benchmark collection has been defined, the first analysis that is going to be carried out is the study of the transformation selected for each benchmark and problem size. The objective of this analysis is proving correct the hypothesis of the advantages a multiversioning system may have, selecting different transformations depending on the characteristics both of the benchmark and of the platform upon which it is being executed. This analysis will be presented in Subsubsection 8.2.1.2.
- **Performance evaluation:** To conclude this section, the last analysis will be the evaluation of the performance achieved with the multiversioning system with respect to a standard compiler –i.e., gcc– and to the original version of Apollo, hereafter referred to as *standard Apollo*. The performance will be assessed both through the speedup and the number of iterations needed to overcome the overhead added by the multiversioning system. This evaluation will be introduced in Subsubsection 8.2.1.3.

8.2.1.1 Specific setup

As mentioned before, this section is going to conclude the refinement of the benchmark suites that are going to be used, checking if Apollo is successful in finding an optimizing transformation; if not, the rationale behind it is going to be assessed.

For instance, for the smallest problem sizes of the benchmarks, Apollo is expected to not provide any performance improvement; as already mentioned, the instrumentation and transformation selection phases of Apollo introduce an overhead that, for small kernels, can cause the original thread to finish before the transformation is computed. In those cases, the multiversioning system can provide better results, since, as presented in Chapter 6, this overhead-masking mechanism has been modified so as to allow Apollo to spend more time in generating the transformation. Hence, these benchmarks must be evaluated, since they can assess some of the new functionalities of the multiversioning system and highlight some of its advantages.

On the contrary, there may be cases in which neither Apollo nor the multiversioning system are able to provide any improvement with respect to the original code. This is the case of tests in which, even with the extensions of Apollo to the polyhedral model,

this model is still not amenable to those codes. A clear example of this behavior are codes in which data dependences are unavoidable, thus preventing transformations to be successful. Maintaining these kernels within the collection of benchmarks used to evaluate the multiversioning system will not provide any useful information, since they are out of the scope of this system. Consequently, those benchmarks belonging to this category are going to be removed from the benchmark suites.

- **PolyBench:** To take the final decision of whether to include all the PolyBench kernels –presented in Subsection 7.1.2 and summarized in Table 7-1– in the functional validation of the multiversioning system or not, they are going to be categorized into three different groups: tests in which Apollo is not even able to compile the code, tests in which Apollo is not able to apply the polyhedral model at runtime, and tests in which Apollo presents a normal behavior.
 - **Compiling issues:** In this case, 5 kernels present compiling-related issues which prevent Apollo from being successful in generating the executable. Specifically, they involve floating-point scalars, which are currently not supported by Apollo. Nonetheless, this is a known issue in Apollo, and there is an ongoing work to solve it. These tests are `symm`, `durbin`, `gramschmidt`, `ludcmp` and `deriche`. Since this behavior cannot be modified by the proposed system, these kernels have been removed from the final set.
 - **Strange behavior:** Another 5 kernels present a strange behavior once compiled and executed with Apollo. First, the LLVM representation of `gemver`, `atax` and `bicg` extracted in the compiling phase presents empty SCoPs. As already described, a SCoP is a part of the code that is compliant with the polyhedral model; if empty, the polyhedral model cannot be applied, and thus Apollo runtime is not able to perform any transformation. Secondly, `lu` and `nussinov` present some unavoidable dependences, so Apollo cannot generate a successful optimizing transformation. As happened before, these issues are out of the scope of the multiversioning system, so they have been also removed from the final set.
 - **Normal behavior:** A total of 20 out of the 30 kernels composing PolyBench can be executed normally with Apollo, which are the ones that will be conforming the final set of PolyBench used in this chapter to address the functional validation of the multiversioning system.
- **ApolloBench:** As stated in [Caamaño’16], two ApolloBench kernels –presented in Subsection 7.1.2 and summarized in Table 7-3– present scenarios where Apollo fails to optimize the code, `spmatmat-rdm` and `spmatmat-worst`. These benchmarks perform a matrix multiplication in which inputs contain memory addresses randomly generated; with this behavior, neither Apollo nor the multiversioning system will ever be able to provide any improvement, since random memory accesses imply that no prediction model can be ever built. Additionally, another version of `spmatmat`, `spmatmat-tube`, contains the so-called *tubes* –memory accesses that cannot be linearly modeled–, and Apollo

is again not able to build a prediction model. Consequently, these three benchmarks have been removed, keeping 9 kernels within ApolloBench.

Hence, the benchmark suite with which the polyhedral extensions are going to be validated comprises 20 kernels from PolyBench and 9 from ApolloBench. Nonetheless, they have had to be adapted to make them amenable to the multiversioning system before using them for validation purposes. As the multiversioning system is based on invoking the same kernel iteratively, all the benchmarks have been adapted to be compliant with this behavior –in other words, to simulate a dataflow-based application–. Since the transformation set is composed of 15 different transformations, each kernel is going to be launched iteratively 30 times, so half of them can be invested in the training phase –from 1 to 15, each iteration testing a different transformation from the set– and the other half in the operational one –from 16 to 30, all of them directly launching the selected best version–. It should be noted that the validation related to the existence of dynamic parameters will not be addressed in this study, but in the one presented in Subsection 8.2.2.

Regarding the working modes, as mentioned at the end of Section 8.1, two of them will be used in the experiments presented in the remaining of this section. The rationale behind this is that, by doing so, Apollo can configure the amount of time invested in generating the optimizing transformation to test.

Specifically, the two working modes that are going to be used in this subsection are modes 2 and 3: the former regulates the time invested in generating the transformed code through a relative timeout, while the latter establishes an absolute one. These two modes are going to be used to devise two different scenarios:

- **2×**: This scenario uses mode 2 to establish a timeout equal to twice the time required by the sequential thread within Apollo to complete the execution of the kernel. In this mode, Apollo launches the sequential thread devised as a mechanism to mask Pluto's overhead, but, if it finishes before Pluto, Apollo waits an amount of time equal to the time elapsed until that moment before retrieving the result of this sequential thread and finishing the computation. In this way, Pluto has an extra time to generate the transformed code; if this extra time is still not enough, then Apollo stops the execution and provides the results computed by the sequential thread. Although this mode adds an extra overhead, it can be canceled out by the multiversioning system in the operating phase. This scenario intends to represent those applications with strict performance requirements, which cannot afford having a large timeout.
- **∞**: This scenario uses mode 3 to establish such a large timeout that it can be considered infinite. Although this is not a realistic scenario, it is proposed so as to evaluate the existing tradeoff between the overhead added by Pluto and the potential benefit it can have of the final performance improvement. In this mode, Pluto has always enough time to generate the transformed code, so all the transformations are evaluated, avoiding missing a potential best performing transformation because it needs a long time to be generated.

Finally, regarding the problem sizes presented in Subsection 7.1.2, PolyBench presented 5 different sizes –MINI, SMALL, MEDIUM, LARGE and EXTRALARGE–, while ApolloBench presented only 3 –SMALL, MEDIUM and LARGE–. As the objective of the smaller sizes is to assess if the multiversioning system is successful were standard Apollo failed and the SMALL problem size is sufficiently small, the MINI problem size has been removed from the set of experiments.

To sum up, the experiments have been configured as follows: first, each benchmark has been modified to invoke 30 times the same kernel –understanding kernel as target loop nest– for each of the predefined problem sizes; then, each modified benchmark has been executed both in Magerit and Namaka using mode 2, following the configuration presented in Section 8.1; to ensure the accuracy of the results, the same experiments have been relaunched 5 times, being the results provided the average of those executions. Finally, the whole process has been repeated for mode 3.

The remainder of the subsection is organized as follows: first, Subsubsection 8.2.1.2 is going to present a study of the transformation selected for each benchmark, problem size and hardware platform; afterwards, Subsubsection 8.2.1.3 is going to evaluate these benchmarks in terms of performance.

8.2.1.2 Transformation selection

One of the main objectives of this subsection is to validate the multiversioning system by showing that, under different circumstances, the transformation that best fits each kernel varies, hence proving correct the hypothesis of the benefits such a system may offer. To do so, a thorough study of the transformation selected for each benchmark, problem size, hardware platform and mode is addressed. The obtained results are summarized in Table 8-4 –mode 2– and Table 8-5 –mode 3–. In both tables, the information is presented following the same structure: the first columns represent the different benchmarks evaluated, starting with ApolloBench and ending with PolyBench; the third column denotes the identifier of the loop nest being processed by Apollo within each benchmark –since more than one can be processed simultaneously–; finally, the rest of the columns provide the transformation selected in each architecture for every problem size (S=Small, M=Medium, L=Large, E=Extralarge), using to that end the transformation ID shown in Table 8-2.

Additionally, Table 8-6 presents a summary of the number of times each transformation has been selected depending on the target platform and the execution mode. This table has been extracted by adding up all the times each transformation is selected, regardless of the specific benchmark. The *Total* column has been computed by adding up all the times each transformation has been selected –i.e., adding up each row–, while the *Total loop kernels* row has been computed by adding up the number of loop kernels evaluated for each mode –i.e., adding up each column–. As can be observed, green and red colors have been used to represent the most and least selected transformations, respectively, while yellow has been used to highlight those cases in which a transformation is chosen many times in one architecture, but not in the other.

Table 8-4: Optimizations selected by the multiversioning system configured with mode 2, establishing a timeout of $2\times$ the sequential time (S=Small, M=Medium, L=Large, E=Extralarge).

Suite	Program	#Loop nest	Namaka				Magerit			
			S	M	L	E	S	M	L	E
ApolloBench	backprop	1	0	3	3	–	1	2	2	–
	dmatmat	1	1	1	8	–	1	1	12	–
	ispmatmat	1	0	0	8	–	0	1	12	–
	mri-q	1	3	14	–	–	3	11	–	–
	needle	1	0	0	7	–	0	0	8	–
		2	0	0	14	–	0	0	9	–
	pcg	1	0	0	9	–	1	3	7	–
	sor	1	0	0	6	–	0	0	6	–
	spmatmat-diag	1	0	3	4	–	0	1	1	–
	spmatmat-square	1	1	1	7	–	1	1	11	–
PolyBench	correlation	1	0	0	0	8	0	0	12	6
		2	0	0	0	1	0	0	0	0
		3	0	0	6	1	0	0	0	0
		4	0	0	0	1	0	0	0	0
	covariance	1	0	0	2	8	0	0	12	6
		2	0	0	0	1	0	0	0	0
		3	0	0	0	0	0	0	0	0
	gemm	1	0	0	4	6	0	0	2	2
	gesummv	1	0	0	0	1	0	0	0	0
	syr2k	1	0	0	6	6	0	0	6	12
	syrk	1	0	0	3	6	0	0	6	10
	trmm	1	0	0	10	12	0	0	6	12
	2mm	1	0	2	2	6	0	0	2	3
		2	0	2	2	6	0	0	2	2
	3mm	1	0	0	3	10	0	2	2	2
		2	0	2	3	10	0	0	3	6
		3	0	2	6	6	0	0	2	6
	doitgen	1	0	0	5	5	0	0	5	5
	mvt	1	0	0	1	3	0	0	0	1
		2	0	0	0	1	0	0	0	1
	cholesky	1	0	0	0	0	0	0	0	0
	trisolv	1	0	0	0	0	0	0	0	0
	floyd-warshall	1	0	0	14	14	1	0	14	14
	adi	1	5	5	5	5	0	1	5	5
	fdtd-2d	1	0	0	7	9	0	0	7	8
	heat-3d	1	0	6	1	1	0	0	1	1
	jacobi-1d	1	0	0	0	0	0	0	0	0
	jacobi-2d	1	0	0	4	14	0	0	12	1
	seidel-2d	1	3	2	6	6	0	2	6	6

Table 8-5: Optimizations selected by the multiversioning system configured with mode 3, establishing an infinite timeout (S=Small, M=Medium, L=Large, E=Extralarge).

Suite	Program	#Loop nest	Namaka				Magerit			
			S	M	L	E	S	M	L	E
ApolloBench	backprop	1	3	3	10	–	3	3	3	–
	dmatmat	1	1	1	8	–	1	1	8	–
	ispmatmat	1	0	0	8	–	0	1	8	–
	mri-q	1	14	14	–	–	10	7	–	–
	needle	1	0	8	7	–	0	0	8	–
		2	0	8	14	–	0	0	9	–
	pcg	1	9	9	14	–	7	7	7	–
	sor	1	0	6	6	–	0	0	6	–
	spmatmat-diag	1	0	6	4	–	0	1	1	–
	spmatmat-square	1	1	1	7	–	1	1	1	–
PolyBench	correlation	1	0	6	6	8	0	0	2	6
		2	0	0	4	12	0	12	3	3
		3	0	0	3	3	12	0	3	6
		4	0	0	12	4	0	0	4	12
	covariance	1	12	6	2	8	12	6	2	6
		2	0	0	3	3	0	0	3	3
		3	0	0	3	4	0	0	3	12
	gemm	1	0	2	10	11	0	2	2	7
	gesummv	1	0	4	4	1	0	0	3	6
	syr2k	1	8	10	6	6	5	6	6	10
	syrk	1	0	4	6	5	0	6	1	7
	trmm	1	0	6	12	12	0	6	12	12
	2mm	1	1	6	4	6	0	6	2	6
		2	0	12	11	6	0	2	2	6
	3mm	1	0	12	3	6	0	6	2	6
		2	0	3	10	6	0	12	6	6
		3	0	12	3	10	12	6	6	6
	doitgen	1	0	0	5	5	0	0	5	5
	mvt	1	0	0	4	6	0	8	12	12
		2	0	0	10	10	0	3	4	12
	cholesky	1	0	0	0	0	0	0	0	0
	trisolv	1	0	0	0	0	0	0	0	0
	floyd-warshall	1	0	0	14	14	2	0	14	14
	adi	1	5	5	5	5	5	5	5	5
	fdtd-2d	1	0	0	7	11	0	0	7	8
	heat-3d	1	0	6	1	1	0	12	1	1
	jacobi-1d	1	0	0	0	0	0	0	0	0
	jacobi-2d	1	0	0	3	14	0	0	3	12
	seidel-2d	1	2	2	6	6	2	2	6	6

Table 8-6: Summary of the number of times each version is selected.

ID	Flag combination	Namaka		Magerit		Total
		2×	∞	2×	∞	
0	no transformation	75	50	82	49	256
1	identity	13	8	16	11	48
2	intratile	8	4	12	11	35
3	tile $32 \times 32 \times 32$ + intratile	9	11	4	12	36
4	tile $32 \times 32 \times 32$ + intratile + unroll $\times 2$	3	9	0	2	14
5	tile $32 \times 32 \times 32$ + intratile + unroll $\times 8$	6	7	4	7	24
6	tile $16 \times 16 \times 16$ + intratile	13	19	10	22	64
7	tile $64 \times 64 \times 64$ + intratile	3	3	2	7	15
8	tile $128 \times 128 \times 128$ + intratile	4	7	2	5	18
9	tile $256 \times 256 \times 256$ + intratile	2	2	1	1	6
10	tile $32 \times 32 \times 64$ + intratile	3	7	1	2	13
11	tile $64 \times 64 \times 32$ + intratile	0	3	2	0	5
12	tile $16 \times 64 \times 32$ + intratile	1	8	7	14	30
13	diamond-tile + intratile	0	0	0	0	0
14	tile $32 \times 32 \times 32$ + intratile + l2tile	5	7	2	2	16
Total loop kernels		145	145	145	145	580

First, it should be highlighted that, as expected, the transformation selected as the most suitable one is different depending on the benchmark, the problem size and the target architecture, proving correct the initial hypothesis. Nonetheless, many other interesting conclusions can be extracted from the analysis of Table 8-4:

- For Small and Medium sizes, the most selected version is the original one, regardless of the benchmark suite and the target architecture: out of 39 loop nests, 34 –Namaka– and 33 –Magerit– choose the original version for the Small size; likewise, the numbers for Medium are 27 –Namaka– and 29 –Magerit–. This is also coherent with the initial hypothesis: the workloads of these problem sizes are still too small for the multiversioning system to have enough time to generate and test optimizing transformations and, even if it does have time to generate a transformation, most times the original code is so fast that the transformation does not perform better, thus resulting in not being elected.
- On the contrary, for Large and Extralarge sizes, usually one of the transformations evaluated is selected as the best performing one in detriment of the original version, as expected.
- Regarding the transformations that are chosen in each case, several interesting features can be highlighted from Table 8-6:
 - As can be observed, the original version is clearly the one chosen the most, as it is selected 256 times out of the 580 experiments. This is consistent with the obtained results, since, as already explained, Small and Medium sizes almost always choose this version.

- Surprisingly, the second most-selected transformation is transformation 6, which implements a homogeneous tiling of size 16 and is selected 64 times. This result is especially interesting, as it highlights the benefits of a multiversioning system as the one proposed in this document: although Pluto applies a default tiling size of 32, this test shows that, for the benchmarks evaluated here, a size of 16 throws better performing results.
 - As shown in Table 8-6, transformation 1 is the third most-selected transformation, as it is chosen 48 times. This is also an expected outcome, as this transformation was chosen because it is the one with which Pluto introduces less overhead; hence, it was expected to provide good results when the timeout is tight.
 - The other most-selected transformations are 2 and 3 –35 and 36 times, respectively–, which was also expected: transformation 2 allows Pluto to apply optimizing transformations such as loop interchange, while transformation 3 implements the tiling defined as default by Pluto.
 - On the contrary, transformation 13 is never selected; this was also expected, since this transformation tries diamond tiling, which can be beneficial for some stencil computations, but in the rest of the cases it is not applicable.
 - Finally, the cases highlighted in yellow represent transformations selected many times in one platform, but not in the other, proving once again the benefits of a multiversioning system by showing that the performance of each transformation highly depends on the target architecture.
- There are some cases in which, regardless of the problem size and the target architecture, no transformation is ever selected, and the original code always remains. This is the case of three tests: `cholesky`, `trisolv` and `jacobi-1d`. This can happen due to three main reasons: (i) that, as mentioned before, the timeout is not large enough to let Pluto generate transformations; (ii) that even for the largest problem sizes, the workload is still too low for the multiversioning system to provide any performance improvement; or (iii) that the set of transformations presented in Table 8-2 is not the most suitable one for these specific tests, hence choosing always the original code. The first reason can be discarded, since this behavior remains when using an almost infinite timeout –as shown in Table 8-5, which will be thoroughly analyzed next–; the other two will be analyzed with the performance results presented in Subsubsection 8.2.1.3.

Additionally, some other interesting conclusions can be extracted from the analysis of Table 8-5 and its comparison with Table 8-4:

- Regarding the experiments for which the original version was always selected as the best performing with mode 2, now it can be observed that, for many of those cases, a different transformation is selected. To simplify the understanding of the table, the cases in which the original version was chosen with mode 2,

but a different transformation is chosen with mode 3, are highlighted in blue. As can be noted, there are many cases, namely in the `Small` problem size, that still maintain the original version as the best performing one; since the reason cannot be a too strict timeout anymore, this behavior is presumably due to a very low workload, which makes impossible to obtain a version that runs even faster than the original code. This hypothesis will be assessed in Subsubsection 8.2.1.3, where the performance is going to be evaluated. On the contrary, for larger problem sizes, it can be observed that the number of times in which the original version is chosen has been drastically reduced. Specifically, for `Extralarge` sizes, it can be seen that only the three cases mentioned before remain with the original version as the best one: `cholesky`, `trisolv` and `jacobi-1d`. As mentioned before, these benchmarks will be further analyzed in Subsubsection 8.2.1.3.

- Regarding the transformations selected for each problem size and target architecture, several conclusions can also be extracted:
 - With respect to transformation 1, Table 8-6 shows that, with mode 3 (∞), this transformation is still chosen, but considerably less times than with mode 2 ($2\times$), as it is reduced from 13 times to 8 for `Namaka` and from 16 to 11 for `Magerit`. This is also coherent with the initial hypothesis: this transformation guarantees a minimal overhead, since `Pluto` does not perform any transformation, but only checks if the loop can be parallelized. Consequently, this transformation was expected to be chosen many times when fixing a tight timeout, but it seems also logic that, when this timeout is not that tight, other transformations may be more powerful.
 - As happened before, transformation 13 is not chosen, proving that, even if powerful for some stencil computations, its applicability may be restricted.
 - With respect to the rest of the transformations, it can be observed that, even if a transformation was already chosen with mode 2, the selection may have changed with mode 3. The rationale behind this can be twofold: (i) that the transformation chosen with mode 3 is the best performing one, but there was not enough time to compute and test it with mode 2, hence choosing the best one among those already computed with the timeout of $2\times$; or (ii) that those specific transformations perform similarly, hence being a matter of a few decimals which one is being chosen. To discern if the reason is the former or the latter, the performance of each one must be analyzed, which will be done in Subsubsection 8.2.1.3.

To close this study, the conclusions extracted from Table 8-4, Table 8-5 and Table 8-6 prove correct two of the initial hypotheses from which the work presented in this document started:

- It has been proven that, depending on features such as the problem size, the target architecture or the kernel under study, a different transformation may

provide the best results in terms of performance, thus validating the hypothesis that a multiversioning system may be beneficial for dataflow-based applications.

- It has been also proven that, when granted enough time, it is possible for Pluto to provide better performing transformations than those obtained when putting strict limitations to the overhead Pluto may add; consequently, a tradeoff between the overhead that can be considered acceptable and the performance of the optimizing transformations must be established.

To conclude the functional validation of the extensions added to Apollo, the remainder of this section is going to assess the performance of the transformations selected in Table 8-4 and Table 8-5.

8.2.1.3 Performance evaluation

Once the transformations chosen for each experiment have been studied, this section deals with the performance characterization of the same experiments –hereafter, increasing the performance means reducing the execution time–. To be able to compare all of them, this study is organized as follows: first, the execution times considered as the baselines for each architecture to assess the performance will be presented; afterwards, the execution time of the multiversioning system will be characterized. To that end, this time is going to be divided into two different categories, since the time spent in each phase of the multiversioning system –training phase and operational phase– will be measured separately. These times will serve as a basis to study how the multiversioning system behaves in terms of performance.

The baseline times are presented in Table 8-7 and Table 8-8; the former gathers the execution time of the different benchmarks measured for Clang; the latter provides the execution time measured for the original version of Apollo, prior to any extension of its dynamic phase –hereafter, referred to as standard Apollo–. From these tables, two important features can be extracted:

- First, with respect to Clang times, it can be observed that, in general lines, the execution time of the benchmarks for the `Small` problem size is already too low: as can be seen in Table 8-7, the tests with execution times below 1 ms have been highlighted in orange; for the `Small` problem size, it can be seen that 19 out of 29 loop nests are highlighted for both architectures. This is coherent with the results obtained in the transformation selection study, where an optimizing transformation was rarely selected over the original sequential time.
- Secondly, regarding the performance achieved with the original version of Apollo, it can be seen that, in general lines, Apollo fails to provide any speedup for the smallest problem sizes –namely, `Small` and `Medium`–; on the contrary, it can reach high speedups for the `ExtraLarge` problem size: for instance, covariance executed with Clang takes more than 62 seconds in Namaka, while with Apollo

Table 8-7: Performance results obtained with Clang (in seconds).

Program	Namaka				Magerit			
	S	M	L	E	S	M	L	E
backprop	0.020	1.051	10.178	–	0.037	0.511	12.846	–
dmatmat	0.082	0.860	11.880	–	0.094	0.857	21.378	–
ispmatmat	0.087	0.871	11.917	–	0.103	0.892	21.855	–
mri-q	1.566	8.173	–	–	1.738	9.257	–	–
needle	1.7×10^{-4}	0.046	17.834	–	4.1×10^{-4}	0.020	39.368	–
pcg	0.233	0.792	7.999	–	1.136	3.912	40.083	–
sor	3.7×10^{-5}	0.029	30.047	–	5.5×10^{-5}	0.037	37.731	–
spmatmat-diag	0.022	1.050	13.500	–	0.024	1.270	18.231	–
spmatmat-square	0.221	0.614	8.149	–	0.211	1.075	12.570	–
correlation	2.3×10^{-4}	0.007	4.274	62.085	3.3×10^{-4}	0.008	1.305	16.438
covariance	2.3×10^{-4}	0.007	4.228	62.078	3.6×10^{-4}	0.009	1.353	16.651
gemm	7.9×10^{-5}	0.002	0.500	4.792	9.8×10^{-5}	0.002	0.398	6.309
gesummv	3.7×10^{-5}	2.8×10^{-4}	0.008	0.035	2.0×10^{-5}	1.4×10^{-4}	0.004	0.020
syr2k	1.2×10^{-4}	0.004	2.853	44.463	1.9×10^{-4}	0.005	1.152	17.665
syrk	5.7×10^{-5}	0.002	0.442	16.630	1.1×10^{-4}	0.003	0.551	6.406
trmm	8.7×10^{-5}	0.003	1.702	32.076	1.2×10^{-4}	0.005	0.785	7.922
2mm	1.9×10^{-4}	0.011	1.447	45.530	2.6×10^{-4}	0.015	1.921	20.038
3mm	3.6×10^{-4}	0.017	2.962	52.927	4.5×10^{-4}	0.022	3.001	33.422
doitgen	3.6×10^{-4}	0.005	0.402	3.111	2.9×10^{-4}	0.006	0.588	4.381
mvt	2.0×10^{-5}	2.4×10^{-4}	0.019	0.084	2.7×10^{-5}	3.6×10^{-4}	0.012	0.062
cholesky	1.9×10^{-4}	0.008	1.258	10.003	2.3×10^{-4}	0.010	1.421	13.020
trisolv	5.6×10^{-6}	6.3×10^{-5}	0.003	0.009	7.9×10^{-6}	9.0×10^{-5}	0.002	0.011
floyd-warshall	0.004	0.079	17.222	128.445	0.003	0.070	14.069	126.930
adi	0.002	0.051	7.794	64.518	0.002	0.069	9.571	85.391
fdtd-2d	1.9×10^{-4}	0.005	2.232	19.054	2.5×10^{-4}	0.006	1.589	21.708
heat-3d	4.2×10^{-4}	0.009	2.327	19.940	4.9×10^{-4}	0.013	1.896	36.557
jacobi-1d	2.8×10^{-6}	2.0×10^{-5}	5.1×10^{-4}	0.002	3.8×10^{-6}	3.7×10^{-5}	3.8×10^{-4}	0.002
jacobi-2d	2.8×10^{-4}	0.006	2.037	18.890	2.9×10^{-4}	0.007	1.616	34.836
seidel-2d	0.004	0.122	15.494	124.236	0.006	0.174	22.333	178.966

Table 8-8: Performance results obtained with standard Apollo (in seconds).

Program	Namaka				Magerit			
	S	M	L	E	S	M	L	E
backprop	0.063	0.135	0.691	–	0.116	0.194	1.477	–
dmatmat	0.164	0.326	1.897	–	0.170	0.843	7.323	–
ispmatmat	0.123	1.843	2.278	–	0.168	0.785	7.469	–
mri-q	0.196	0.711	–	–	0.571	3.034	–	–
needle	0.090	0.133	0.869	–	0.021	0.089	3.628	–
pcg	0.909	1.708	4.947	–	1.893	2.243	7.242	–
sor	0.002	0.088	5.188	–	0.001	0.079	14.193	–
spmatmat-diag	0.107	0.687	5.292	–	0.048	1.248	10.248	–
spmatmat-square	0.305	0.494	2.454	–	0.210	0.732	6.831	–
correlation	0.112	0.396	0.859	2.680	0.015	0.035	0.486	1.510
covariance	0.228	0.092	0.692	2.343	0.007	0.034	0.323	1.435
gemm	0.018	0.076	0.267	1.017	0.004	0.024	0.225	1.129
gesummv	0.021	0.017	0.093	0.122	0.003	0.003	0.025	0.089
syr2k	0.101	0.095	0.368	1.554	0.015	0.029	0.237	1.271
syrk	0.125	0.123	0.262	0.654	0.018	0.025	0.192	0.702
trmm	0.078	0.083	0.597	4.478	0.007	0.019	0.309	1.694
2mm	0.293	0.110	0.680	2.112	0.008	0.053	0.679	1.621
3mm	0.306	0.228	1.066	2.936	0.037	0.071	1.100	2.875
doitgen	0.041	0.043	0.692	3.127	0.026	0.049	0.672	3.255
mvt	0.078	0.044	0.121	0.137	0.003	0.020	0.048	0.083
cholesky	0.062	0.033	3.692	29.872	0.019	0.039	3.183	26.331
trisolv	0.128	0.258	0.140	0.129	0.012	0.013	0.141	0.124
floyd-warshall	0.045	0.163	8.751	93.694	0.016	0.103	4.240	24.744
adi	0.050	0.192	12.980	102.904	0.081	0.147	10.652	89.785
fdtd-2d	0.036	0.042	0.732	19.266	0.011	0.045	0.495	5.773
heat-3d	0.111	0.131	0.588	19.370	0.140	0.167	0.605	5.214
jacobi-1d	0.023	0.051	0.023	0.025	0.003	0.003	0.003	0.016
jacobi-2d	0.030	0.035	0.276	16.079	0.009	0.034	0.318	4.686
seidel-2d	0.031	0.078	1.944	12.466	0.033	0.084	2.201	14.339

this time is reduced to less than 3 seconds. This case has been highlighted in green in Table 8-7 and Table 8-8 to ease the readability of the tables.

- Finally, with respect to the overhead added by Apollo, it can be noticed that, for the smallest problem sizes, this overhead can be extremely large. For instance, `trisolv-Small` has been highlighted in red in Table 8-7 and Table 8-8 to exemplify one of the cases with very large overhead, since Clang takes 5.6 μ s and Apollo requires 0.128s. This behavior can be due to two main reasons:

- **Instrumentation added by Apollo:** When the runtime manager of Apollo triggers the mechanism to mask the overhead, it launches a thread that executes the sequential version of the code while the main thread deals with the transformation generation. However, this sequential version is not the original code, but the one that has already been instrumented by the static part of Apollo. Although this code is obviously semantically correct and equivalent to the original one, it includes some instructions to track memory accesses or the scalar evolution. Consequently, there is an extra overhead added by this version when it finishes before the main thread completes the generation of the transformation. Although this overhead can be considered negligible when the workload of the kernel under analysis is large enough, for extremely low workloads, this assumption does not hold anymore.
- **Implementation of the overhead-masking mechanism:** As mentioned before, the overhead-masking mechanism consists of launching a thread running sequentially the code while the transformation selection stage is taking place. Within Apollo, this mechanism has been implemented as a set of checkpoints: the transformation selection stage is composed of many sequential operations, and before and after each one, the sequential thread is checked, going forward if this thread is still running, and stopping the transformation selection stage if it has already finished. As it is implemented as a set of checkpoints, and not as an interruption, if the sequential thread finishes while one of the operations is being executed, it is not detected until that operation finishes. Hence, the time elapsed between the ending of the sequential thread and the ending of the operation being executed supposes an extra overhead. This overhead can be considered negligible when the workload of the kernel is large enough, but this assumption does not hold anymore for extremely low workloads.

Once the baseline times have been analyzed, the remaining of this section is going to focus on characterizing the performance of the multiversioning system for the two working modes selected, hereafter referred to as $2\times$ and infinite. Before getting into the details of the characterization itself, first the concept of invocation must be clearly stated. In the context of this PhD, an invocation is defined as each time the same kernel is relaunched with the same combination of parameters and, hence, the multiversioning system can be put to work.

To characterize the performance, the following information is gathered:

- **Training time:** For each kernel, problem size and target architecture, this time will represent the average per invocation of the time the multiversioning system invests in the training phase. As a reminder, during the 15 invocations of this phase, 15 different transformations are tested, one per invocation.
- **Operational time:** Likewise, this time gathers the average per invocation of the time the multiversioning system requires to execute each kernel in the operational phase, once a transformation has already been selected and memoized; thus, the transformation is directly retrieved each time the same kernel with the same parameters is revisited.
- **Speedup with respect to Clang / original version of Apollo:** These figures of merit provide the ideal speedup measured with respect to each baseline time. In this context, *ideal speedup* is understood as a case in which the same kernel has been relaunched with the same parameters so many times that the time of the training phase can be considered negligible, thus considering only the time of the operational phase to compute the speedups, as shown in Equation 8-1 and Equation 8-2 for Clang and standard Apollo, respectively. This ideal speedup has been chosen to assess the multiversioning system because it represents the maximum speedup it can ever be achieved with such a system.

$$SpdpC = \frac{Oper}{ClangTime} \quad (8-1)$$

$$SpdpS = \frac{Oper}{ApolloTime} \quad (8-2)$$

- **Invocations to overcome Clang / original version of Apollo:** Since the training phase of the multiversioning system is expected to introduce a non-negligible overhead, the aforementioned speedups can only be understood when they are considered together with this new figure of merit, which represents the number of times the same kernel must be invoked with the same parameters to start providing an acceleration.

This information is gathered in Table 8-9, Table 8-10, Table 8-11 and Table 8-12. Since one of the main objectives is to compare the differences observed when using two different working modes, to ease the readability of this document, these tables group the information by benchmark suite, and not by working mode. Hence, the first two tables provide this information for ApolloBench when the multiversioning system is working in mode 2 –2× timeout– and in mode 3 –infinite timeout–, respectively, while the last two tables present the same information, but for PolyBench. Additionally, to ease the readability of the tables and the understanding of the conclusions extracted from them, a color code has been used to highlight the most relevant results.

In these tables, *Train* represents the average time of the training phase, while *Oper* does the same with the average time of the operational phase; likewise, *SpdpC* provides the speedup achieved with the operational phase when compared to Clang, while *SpdpS* does the same, but when compared with standard Apollo; finally, *InvC* and *InvS* represent the number of invocations needed to overcome Clang and standard Apollo, respectively. With respect to the metrics, all the times are represented in seconds, while the speedups are provided in times.

Before getting to the analysis of the obtained results, the methodology followed to compute the number of invocations to overcome the baseline –either Clang or standard Apollo– must be clearly stated. As aforementioned, this figure of merit represents the number of invocations needed to cancel out the overhead introduced by the training phase; in other words, it establishes in which iteration the time spent by the baseline is equal to or greater than the time spent by the multiversioning system, which is composed by the time spent in the training phase and the time invested in the operational one. This approach is gathered in Equation 8-3, where *N* represents the number of transformations tested in the training phase –in this case, 15.

$$Baseline \cdot \#inv = N \cdot Train + (\#inv - N) \cdot Oper \quad (8-3)$$

By isolating the number of invocations, Equation 8-4 is obtained. As can be seen, the number of invocations can take three different values depending on the specific values of *Oper*, *Train* and *Baseline*, assuming *Oper* is always smaller than *Train*:

- If the average training time per invocation is greater than the baseline, the number of invocations needed to cancel out the overhead is computed following the expression presented in Equation 8-4.
- On the contrary, if the average training time per invocation is already smaller than the baseline, it means that, even when executing the training phase, Apollo provides better performing results. Consequently, this behavior will be denoted with 1, which means that the overhead is canceled out since the beginning. Nonetheless, it must be noted that, since the training time is an average, there can be cases in which the first invocation is not faster than the baseline; as will be analyzed later, the execution time per invocation can present a huge variation during the training phase, since one transformation can be very efficient, but the next can be very inefficient, thus providing very slow execution times.
- Finally, if the average operating time per invocation is greater than or equal to the baseline, this means that, even with the optimizations added by the multiversioning system, the result is slower than the baseline; consequently, computing this number is no longer an option.

$$\#inv = \begin{cases} N \cdot (Train - Oper) / (Baseline - Oper) & \text{if } Train > Baseline \\ 1 & \text{if } Train \leq Baseline \\ - & \text{if } Oper \geq Baseline \end{cases} \quad (8-4)$$

Table 8-9: Performance evaluation of ApolloBench with mode 2 (in seconds), establishing a timeout of $2\times$.

Program		Namaka						Magerit					
		Train	Oper	SpdpC	SpdpS	InvC	InvS	Train	Oper	SpdpC	SpdpS	InvC	InvS
backprop	S	0.813	0.022	0.908	2.851	–	291	0.192	0.005	8.197	25.804	87	26
	M	1.275	0.045	23.393	3.005	19	205	0.780	0.097	5.264	1.993	25	107
	L	4.577	0.469	21.684	1.472	1	279	6.351	0.978	13.141	1.511	1	162
dmatmat	S	0.806	0.030	2.694	5.407	227	88	0.675	0.026	3.673	6.640	143	68
	M	1.375	0.143	6.001	2.273	26	102	1.222	0.132	6.477	6.372	23	24
	L	4.476	1.459	8.142	1.300	1	104	4.769	1.433	14.922	5.111	1	1
ispmatmat	S	0.126	0.115	0.757	1.065	–	22	0.137	0.125	0.825	1.344	–	1
	M	2.983	0.925	0.942	1.993	–	34	1.537	0.157	5.671	4.994	29	33
	L	5.317	1.742	6.840	1.307	1	101	5.652	1.638	13.343	4.560	1	1
mri-q	S	8.464	0.117	13.412	1.678	87	1582	5.721	0.143	12.112	3.980	53	196
	M	43.066	0.606	13.483	1.173	85	6086	28.638	0.754	12.284	4.026	50	184
needle	S	0.562	2.9×10^{-4}	0.579	313.847	–	94	0.006	2.6×10^{-4}	1.594	82.512	517	1
	M	1.150	0.046	1.014	2.920	25605	189	0.394	0.034	0.580	2.619	–	99
	L	9.220	0.757	23.550	1.148	1	1133	9.325	0.244	161.348	14.869	1	41
pcg	S	2.488	0.287	0.810	3.162	–	54	4.483	1.988	0.571	0.952	–	–
	M	7.259	0.864	0.917	1.976	–	114	9.959	0.385	10.167	5.829	41	78
	L	24.300	2.661	3.006	1.859	61	143	33.645	2.355	17.023	3.076	1	97
sor	S	0.001	6.1×10^{-4}	0.061	2.671	–	1	0.001	8.9×10^{-4}	0.061	1.585	–	1
	M	0.327	0.033	0.869	2.678	–	81	0.310	0.051	0.724	1.559	–	137
	L	15.552	2.391	12.567	2.170	1	71	17.220	2.735	13.795	5.189	1	19
spmatmat-diag	S	0.270	0.020	1.084	5.262	2193	44	0.115	0.028	0.872	1.702	–	67
	M	2.820	0.595	1.762	1.154	74	364	2.384	0.504	2.519	2.475	37	38
	L	10.019	3.699	3.650	1.431	1	60	9.020	6.063	3.007	1.690	1	1
spmatmat-square	S	1.476	0.072	3.061	4.223	142	91	1.439	0.037	5.707	5.689	122	122
	M	2.083	0.144	4.260	3.427	62	84	1.873	0.130	8.284	5.639	28	44
	L	5.393	1.358	6.001	1.807	1	56	4.989	1.069	11.764	6.393	1	1

Table 8-10: Performance evaluation of ApolloBench with mode 3 (in seconds), establishing an infinite timeout.

Program		Namaka						Magerit					
		Train	Oper	SpdpC	SpdpS	InvC	InvS	Train	Oper	SpdpC	SpdpS	InvC	InvS
backprop	S	0.607	0.004	4.465	14.016	582	155	0.567	0.005	6.921	21.788	267	76
	M	0.849	0.048	22.067	2.835	1	138	0.723	0.037	13.651	5.170	22	66
	L	4.665	0.450	22.636	1.536	1	263	2.684	0.356	36.046	4.146	1	32
dmatmat	S	0.652	0.030	2.701	5.421	182	70	0.648	0.034	2.748	4.968	155	68
	M	0.942	0.161	5.348	2.025	17	72	0.991	0.215	3.988	3.923	19	19
	L	3.799	1.464	8.115	1.296	1	81	4.526	1.947	10.982	3.762	1	1
ispmatmat	S	0.124	0.113	0.770	1.083	–	17	0.141	0.127	0.814	1.326	–	1
	M	8.596	0.975	0.893	1.889	–	132	1.376	0.279	3.196	2.814	27	33
	L	4.314	1.744	6.835	1.306	1	73	5.369	2.288	9.551	3.264	1	1
mri-q	S	7.943	0.118	13.312	1.666	82	1500	5.678	0.197	8.834	2.903	54	220
	M	41.269	0.615	13.293	1.156	81	6352	29.115	0.971	9.538	3.126	51	205
needle	S	0.880	0.001	0.168	91.094	–	148	0.772	3.7×10^{-4}	1.129	58.461	244416	550
	M	0.917	0.008	6.160	17.737	352	109	0.901	0.034	0.585	2.644	–	236
	L	7.892	0.769	23.183	1.130	1	1068	14.554	0.355	110.884	10.218	1	66
pcg	S	12.721	0.178	1.309	5.109	3424	258	11.985	0.277	4.092	6.823	205	109
	M	13.586	0.359	2.204	4.752	459	148	13.651	0.425	9.213	5.282	57	110
	L	24.444	2.678	2.987	1.847	62	144	34.196	2.331	17.194	3.107	1	98
sor	S	0.001	6.4×10^{-4}	0.058	2.553	–	1	0.001	9.1×10^{-4}	0.060	1.565	–	1
	M	1.733	0.012	2.416	7.450	1547	340	1.684	0.048	0.770	1.657	–	782
	L	14.187	2.392	12.561	2.169	1	64	16.829	3.659	10.311	3.879	1	19
spmatmat-diag	S	2.729	0.020	1.104	5.363	19459	466	2.981	0.029	0.830	1.620	–	2430
	M	4.513	0.491	2.138	1.401	109	307	5.010	0.823	1.544	1.517	141	148
	L	10.330	4.382	3.081	1.208	1	98	12.059	6.564	2.777	1.561	1	23
spmatmat-square	S	1.285	0.072	3.065	4.229	123	79	1.291	0.044	4.795	4.780	113	113
	M	1.622	0.164	3.743	3.011	49	67	1.663	0.194	5.548	3.776	26	41
	L	4.383	1.357	6.006	1.809	1	42	4.682	1.255	10.015	5.443	1	1

Table 8-11: Performance evaluation of PolyBench with mode 2 (in seconds), establishing a timeout of $2 \times$.

Program		Namaka						Magerit					
		Train	Oper	SpdpC	SpdpS	InvC	InvS	Train	Oper	SpdpC	SpdpS	InvC	InvS
correlation	S	0.986	0.001	0.201	99.410	–	133	0.037	0.001	0.223	10.034	–	40
	M	1.443	0.009	0.816	46.153	–	56	0.205	0.013	0.622	2.734	–	131
	L	4.413	0.117	36.678	7.375	16	87	1.704	0.108	12.067	4.496	21	64
	E	10.669	1.378	45.057	1.945	1	108	4.346	1.045	15.725	1.444	1	107
covariance	S	0.679	0.001	0.223	217.103	–	45	0.017	0.001	0.258	5.075	–	41
	M	0.926	0.009	0.750	10.591	–	166	0.175	0.012	0.689	2.688	–	116
	L	3.451	0.120	35.279	5.778	1	88	1.534	0.101	13.360	3.187	18	97
	E	9.391	1.209	51.354	1.938	1	109	4.014	1.013	16.438	1.417	1	107
gemm	S	0.106	5.4×10^{-4}	0.147	33.713	–	90	0.003	0.001	0.099	3.777	–	1
	M	0.121	0.003	0.718	25.320	–	25	0.033	0.003	0.637	8.513	–	22
	L	1.220	0.079	6.334	3.382	41	92	1.071	0.077	5.198	2.933	47	101
	E	2.904	0.731	6.560	1.392	1	114	2.769	0.844	7.478	1.338	1	102
gesummv	S	0.086	3.5×10^{-4}	0.107	59.836	–	63	0.003	2.0×10^{-4}	0.101	13.562	–	1
	M	0.167	5.3×10^{-4}	0.532	32.397	–	150	0.014	3.5×10^{-4}	0.421	8.152	–	82
	L	0.232	0.009	0.807	10.047	–	40	0.069	0.006	0.688	4.235	–	50
	E	0.646	0.006	5.840	20.305	329	83	0.241	0.026	0.756	3.374	–	52
syr2k	S	0.120	7.3×10^{-4}	0.163	137.444	–	18	0.024	0.001	0.181	13.852	–	26
	M	0.336	0.009	0.422	10.231	–	58	0.308	0.018	0.293	1.541	–	434
	L	1.682	0.090	31.725	4.089	1	86	1.213	0.101	11.386	2.342	16	123
	E	6.426	1.022	43.510	1.521	1	153	4.178	1.176	15.023	1.081	1	475
syrk	S	0.146	3.8×10^{-4}	0.152	334.130	–	18	0.004	6.3×10^{-4}	0.179	28.607	–	1
	M	0.205	0.003	0.731	48.891	–	26	0.040	0.003	0.777	7.518	–	26
	L	1.075	0.054	8.135	4.826	40	74	0.953	0.061	9.012	3.131	28	103
	E	3.282	0.655	25.407	0.999	1	–	2.204	0.712	8.992	0.986	1	–
trmm	S	0.145	5.7×10^{-4}	0.153	136.227	–	29	0.018	0.001	0.124	6.883	–	46
	M	0.170	0.005	0.733	17.660	–	32	0.046	0.006	0.809	3.123	–	48
	L	4.670	0.332	5.127	1.800	48	246	4.025	0.174	4.509	1.777	95	428
	E	10.373	3.128	10.254	1.431	1	81	7.637	1.195	6.630	1.417	1	194

Continued on next page

Table 8-11 – continued from previous page

Program		Namaka						Magerit					
		Train	Oper	SpdpC	SpdpS	InvC	InvS	Train	Oper	SpdpC	SpdpS	InvC	InvS
2mm	S	0.236	0.001	0.191	287.879	–	1	0.034	0.002	0.142	4.220	–	82
	M	0.364	0.002	4.821	49.351	639	51	0.109	0.022	0.667	2.409	–	43
	L	3.429	0.117	12.373	5.814	38	89	3.052	0.091	21.110	7.462	25	76
	E	8.392	0.778	58.512	2.714	1	86	5.507	0.888	22.577	1.827	1	95
3mm	S	0.380	0.002	0.239	200.945	–	19	0.047	0.003	0.168	13.662	–	20
	M	0.546	0.009	1.914	25.670	996	37	0.207	0.025	0.901	2.868	–	60
	L	5.655	0.138	21.434	7.717	30	90	4.712	0.198	15.178	5.565	25	76
	E	13.510	1.122	47.181	2.617	1	103	8.927	1.166	28.661	2.465	1	69
doitgen	S	0.098	0.007	0.052	5.953	–	40	0.048	0.011	0.026	2.319	–	37
	M	0.199	0.012	0.416	3.533	–	92	0.066	0.017	0.334	2.932	–	24
	L	3.087	0.267	1.506	2.593	313	100	2.480	0.371	1.584	1.808	146	106
	E	18.760	1.287	2.416	2.429	144	143	17.650	1.709	2.564	1.905	90	155
mvt	S	0.297	2.8×10^{-4}	0.071	280.877	–	58	0.003	2.2×10^{-4}	0.122	13.105	–	1
	M	0.477	5.7×10^{-4}	0.419	77.933	–	164	0.017	6.5×10^{-4}	0.531	31.075	–	1
	L	0.595	0.007	2.697	17.058	736	78	0.109	0.016	0.756	2.964	–	44
	E	2.685	0.016	5.216	8.482	588	332	2.915	0.010	6.199	8.288	834	595
cholesky	S	0.065	4.1×10^{-4}	0.475	153.408	–	16	0.060	4.9×10^{-4}	0.466	39.302	–	48
	M	0.083	0.008	0.937	3.961	–	46	0.061	0.011	0.945	3.589	–	27
	L	3.232	1.270	0.991	2.908	–	1	2.720	1.438	0.988	2.213	–	1
	E	28.196	10.053	0.995	2.972	–	1	24.809	13.058	0.997	2.017	–	1
trisolv	S	0.175	1.8×10^{-4}	0.032	725.475	–	21	0.003	1.5×10^{-4}	0.051	75.970	–	1
	M	0.153	2.7×10^{-4}	0.238	969.785	–	1	0.032	2.7×10^{-4}	0.331	46.181	–	39
	L	0.199	0.002	0.988	63.987	–	22	0.034	0.003	0.735	47.584	–	1
	E	0.239	0.008	0.992	16.965	–	29	0.195	0.011	1.010	11.477	24970	25
floyd-warshall	S	0.117	0.005	0.772	9.286	–	42	0.075	0.005	0.655	3.209	–	94
	M	0.363	0.096	0.826	1.705	–	60	0.292	0.096	0.727	1.074	–	417
	L	67.320	8.238	2.091	1.062	99	1730	55.340	6.831	2.060	1.060	101	1782
	E	474.218	84.335	1.523	1.111	133	625	535.827	34.202	3.711	1.045	82	4882

Continued on next page

Table 8-11 – continued from previous page

Program		Namaka						Magerit					
		Train	Oper	SpdpC	SpdpS	InvC	InvS	Train	Oper	SpdpC	SpdpS	InvC	InvS
adi	S	1.378	0.007	0.259	6.883	–	483	0.134	0.008	0.271	9.784	–	26
	M	1.503	0.065	0.787	2.943	–	170	0.236	0.089	0.770	1.655	–	38
	L	14.001	8.614	0.905	1.507	–	19	10.526	8.915	1.074	1.195	37	1
	E	101.065	69.871	0.923	1.473	–	1	89.833	78.404	1.089	1.145	25	16
fdtd-2d	S	1.535	0.002	0.079	15.179	–	687	0.032	0.004	0.070	2.986	–	61
	M	1.642	0.011	0.430	3.756	–	797	0.071	0.013	0.477	3.438	–	27
	L	3.732	0.299	7.460	2.447	27	119	3.950	0.294	5.405	1.683	43	274
	E	23.397	18.948	1.006	1.017	631	210	15.507	4.753	4.567	1.215	1	159
heat-3d	S	1.013	0.025	0.017	4.375	–	174	0.091	0.045	0.011	3.110	–	1
	M	1.348	0.035	0.260	3.742	–	206	0.295	0.064	0.200	2.607	–	34
	L	4.186	0.399	5.838	1.476	30	300	3.191	0.420	4.510	1.440	29	225
	E	30.225	17.897	1.114	1.082	91	126	21.292	4.703	7.773	1.109	1	488
jacobi-1d	S	0.098	1.8×10^{-4}	0.015	127.103	–	64	0.002	2.1×10^{-4}	0.018	13.354	–	1
	M	0.119	2.8×10^{-4}	0.073	185.931	–	35	0.002	2.4×10^{-4}	0.159	12.349	–	1
	L	0.105	8.7×10^{-4}	0.590	26.345	–	71	0.011	5.0×10^{-4}	0.755	5.854	–	66
	E	0.106	0.003	0.808	9.020	–	71	0.025	0.002	0.995	8.401	–	24
jacobi-2d	S	0.624	0.002	0.155	16.791	–	328	0.020	0.003	0.101	3.208	–	41
	M	0.698	0.008	0.704	4.294	–	385	0.043	0.008	0.839	4.132	–	21
	L	2.145	0.226	9.015	1.223	16	572	2.688	0.255	6.337	1.249	27	576
	E	18.930	16.051	1.177	1.002	16	1579	12.268	4.283	8.134	1.094	1	298
seidel-2d	S	0.226	0.003	1.421	10.884	2761	118	0.052	0.011	0.538	3.040	–	29
	M	0.637	0.013	9.514	6.135	86	143	0.714	0.017	10.404	5.010	67	156
	L	3.501	1.152	13.455	1.688	1	45	4.272	1.072	20.833	2.053	1	43
	E	20.811	9.038	13.745	1.379	1	52	24.812	7.745	23.107	1.851	1	39

Table 8-12: Performance evaluation of PolyBench with mode 3 (in seconds), establishing an infinite timeout.

Program		Namaka						Magerit					
		Train	Oper	SpdpC	SpdpS	InvC	InvS	Train	Oper	SpdpC	SpdpS	InvC	InvS
correlation	S	2.229	0.001	0.179	88.184	–	302	2.309	0.002	0.175	7.852	–	2640
	M	2.793	0.003	2.546	144.013	9842	107	3.168	0.005	1.438	6.323	19751	1626
	L	3.297	0.122	35.172	7.072	1	65	3.423	0.095	13.732	5.117	42	128
	E	8.876	1.279	48.529	2.095	1	82	5.620	0.986	16.679	1.532	1	133
covariance	S	1.399	0.001	0.247	240.314	–	93	1.532	0.002	0.206	4.048	–	4310
	M	1.858	0.002	3.793	53.587	5804	309	2.148	0.003	3.053	11.904	5563	1048
	L	2.354	0.149	28.376	4.647	1	61	2.397	0.082	16.568	3.952	28	145
	E	7.861	1.190	52.149	4.647	1	87	4.572	0.974	17.101	1.474	1	117
gemm	S	0.609	6.2×10^{-4}	0.127	29.258	–	520	0.688	0.001	0.078	2.968	–	4147
	M	0.687	0.001	1.491	52.580	14565	139	0.795	0.002	0.836	11.166	–	543
	L	0.867	0.064	7.784	4.157	28	60	0.994	0.072	5.513	3.111	43	91
	E	2.228	0.641	7.481	1.587	1	64	2.568	0.826	7.638	1.367	1	87
gesummv	S	0.754	3.0×10^{-4}	0.125	70.253	–	549	0.861	3.8×10^{-4}	0.053	7.129	–	5511
	M	0.848	4.2×10^{-4}	0.679	41.383	–	758	0.990	5.7×10^{-4}	0.259	5.010	–	6544
	L	0.901	0.001	6.129	76.288	2152	147	1.054	0.001	2.970	18.295	5909	673
	E	0.864	0.006	5.855	20.356	441	111	0.983	0.002	8.337	37.214	838	170
syr2k	S	0.755	7.2×10^{-4}	0.166	140.592	–	113	0.795	0.001	0.144	11.024	–	890
	M	4.389	0.003	1.287	31.198	75350	717	5.076	0.004	1.437	7.566	46234	3077
	L	1.126	0.090	31.735	4.090	1	56	1.050	0.100	11.573	2.381	1	104
	E	5.854	1.023	43.483	1.520	1	137	3.919	1.054	16.764	1.206	1	199
syrk	S	0.814	5.6×10^{-4}	0.101	221.880	–	98	0.807	0.001	0.124	19.819	–	710
	M	3.427	0.002	0.751	50.209	–	428	4.869	0.003	0.995	9.626	–	3259
	L	0.756	0.057	7.738	4.591	28	52	0.873	0.093	5.923	2.058	26	119
	E	2.747	0.606	27.444	1.079	1	670	1.922	0.631	10.155	1.114	1	271
trmm	S	1.293	6.2×10^{-4}	0.140	124.603	–	252	1.183	0.001	0.095	5.259	–	3286
	M	1.543	0.004	0.903	21.774	–	292	1.569	0.005	1.046	4.038	110131	1675
	L	2.859	0.237	7.169	2.517	27	110	3.194	0.165	4.760	1.875	74	315
	E	10.002	3.126	10.260	1.432	1	77	7.924	1.198	6.612	1.414	16	204

Continued on next page

Table 8-12 – continued from previous page

Program		Namaka						Magerit					
		Train	Oper	SpdpC	SpdpS	InvC	InvS	Train	Oper	SpdpC	SpdpS	InvC	InvS
2mm	S	1.552	0.001	0.157	236.448	–	80	1.770	0.002	0.107	3.182	–	5030
	M	1.733	0.002	4.693	48.035	3075	242	2.044	0.004	3.809	13.754	2839	626
	L	2.188	0.110	13.170	6.189	24	55	2.648	0.087	22.038	7.790	21	65
	E	6.630	0.769	59.222	2.747	1	66	4.893	0.689	29.101	2.355	1	68
3mm	S	2.435	0.002	0.206	173.068	–	120	2.756	0.003	0.134	10.854	–	1235
	M	2.737	0.004	4.769	63.975	3059	184	3.308	0.005	4.182	13.319	2925	756
	L	3.533	0.132	22.500	8.100	19	55	4.170	0.142	21.090	7.732	22	64
	E	10.448	1.044	50.700	2.812	1	75	8.122	1.079	30.964	2.664	1	59
doitgen	S	0.456	0.007	0.052	5.961	–	196	0.568	0.013	0.023	2.062	–	617
	M	0.849	0.012	0.414	3.519	–	409	0.881	0.021	0.263	2.306	–	470
	L	5.709	0.267	1.508	2.596	603	192	4.585	0.372	1.581	1.805	293	211
	E	22.555	1.273	2.444	2.457	174	173	20.140	1.707	2.567	1.907	104	179
mvt	S	0.301	3.1×10^{-4}	0.064	251.469	–	58	0.284	4.1×10^{-4}	0.066	7.069	–	1707
	M	0.494	5.8×10^{-4}	0.413	76.775	–	170	0.533	6.3×10^{-4}	0.552	32.316	–	408
	L	1.132	0.003	6.116	38.683	1063	145	1.381	0.002	6.748	26.448	1972	446
	E	3.208	0.015	5.728	9.315	689	392	4.863	0.005	12.555	16.785	1271	931
cholesky	S	0.346	5.0×10^{-4}	0.387	125.107	–	85	0.299	6.1×10^{-4}	0.375	31.631	–	240
	M	0.375	0.008	0.924	3.906	–	224	0.319	0.011	0.950	3.611	–	164
	L	3.916	1.276	0.986	2.894	–	17	3.334	1.413	1.005	2.253	3780	17
	E	28.896	10.085	0.992	2.962	–	1	24.974	12.893	1.010	2.042	1420	1
trisolv	S	22.050	2.5×10^{-4}	0.023	512.287	–	2593	25.785	3.5×10^{-4}	0.023	33.876	–	33930
	M	64.872	3.1×10^{-4}	0.205	837.570	–	3773	52.097	3.9×10^{-4}	0.232	32.472	–	63788
	L	54.578	0.003	0.997	57.934	–	5959	41.072	0.003	0.633	40.997	–	4484
	E	53.103	0.009	0.999	16.514	–	6577	39.597	0.012	0.937	10.650	–	5294
floyd-warshall	S	0.220	0.005	0.779	9.364	–	80	0.318	0.007	0.486	2.385	–	497
	M	0.767	0.089	0.887	1.831	–	138	0.717	0.096	0.727	1.074	–	1308
	L	66.489	8.223	2.095	1.064	98	1656	53.231	6.782	2.075	1.068	96	1522
	E	474.812	84.588	1.519	1.108	134	643	535.210	33.341	3.807	1.072	81	3134

Continued on next page

Table 8-12 – continued from previous page

Program		Namaka						Magerit					
		Train	Oper	SpdpC	SpdpS	InvC	InvS	Train	Oper	SpdpC	SpdpS	InvC	InvS
adi	S	1.366	0.007	0.263	6.991	–	478	1.271	0.010	0.219	7.896	–	267
	M	1.512	0.065	0.786	2.942	–	172	1.442	0.072	0.949	2.038	–	274
	L	13.976	8.592	0.907	1.511	–	19	11.780	8.783	1.090	1.213	58	25
	E	101.041	69.914	0.923	1.472	–	1	89.564	77.102	1.108	1.165	23	1
fdtd-2d	S	1.523	0.002	0.081	15.470	–	681	1.543	0.005	0.050	2.146	–	4114
	M	1.650	0.011	0.430	3.757	–	801	1.712	0.022	0.289	2.085	–	1079
	L	3.820	0.289	7.720	2.532	28	120	4.222	0.286	5.551	1.728	46	284
	E	23.794	18.792	1.014	1.025	287	159	14.738	5.060	4.290	1.141	1	204
heat-3d	S	0.989	0.024	0.017	4.522	–	168	0.999	0.050	0.010	2.788	–	158
	M	1.389	0.034	0.264	3.799	–	211	1.371	0.062	0.206	2.679	–	188
	L	4.189	0.398	5.842	1.476	30	300	3.735	0.415	4.570	1.459	34	262
	E	30.294	19.018	1.049	1.019	184	481	21.920	4.751	7.695	1.097	1	557
jacobi-1d	S	0.696	2.6×10^{-4}	0.011	88.630	–	453	0.891	3.5×10^{-4}	0.011	8.086	–	5448
	M	0.792	2.8×10^{-4}	0.071	182.666	–	234	1.052	4.0×10^{-4}	0.093	7.238	–	6293
	L	2.602	8.5×10^{-4}	0.604	26.992	–	1771	3.825	8.6×10^{-4}	0.439	3.408	–	27630
	E	6.172	0.003	0.785	8.759	–	4215	11.655	0.003	0.703	4.822	–	13420
jacobi-2d	S	0.637	0.002	0.159	17.249	–	335	0.674	0.004	0.081	2.563	–	1785
	M	0.696	0.008	0.705	4.302	–	383	0.743	0.012	0.575	2.831	–	503
	L	2.012	0.238	8.551	1.160	1	699	3.051	0.235	6.869	1.354	31	508
	E	19.172	15.948	1.185	1.008	17	371	11.993	4.229	8.238	1.108	1	255
seidel-2d	S	0.365	0.002	2.006	15.360	2659	187	0.472	0.004	1.529	8.630	3486	242
	M	0.435	0.013	9.510	6.132	59	97	0.546	0.017	10.362	4.989	51	119
	L	3.099	1.152	13.452	1.688	1	37	3.820	1.072	20.838	2.054	1	37
	E	20.344	9.043	13.739	1.379	1	50	24.229	7.742	23.116	1.852	1	38

Once the information presented in the tables has been detailed, the most relevant conclusions of this study are going to be extracted. Since there are many cases to be analyzed, first the general conclusions are going to be presented and, afterwards, those cases highlighting interesting features are going to be thoroughly studied. As already mentioned, a color code is used in the tables to guide the analysis of the conclusions.

First, when comparing the results obtained by the multiversioning system with those obtained by the standard version of Apollo, it can be observed that, in general lines, the former outperforms the latter. There are only two exceptions, `pcg-Small` and `syrk-Extralarge` in Magerit –highlighted in orange in the tables–, which fail to outperform standard Apollo in mode 2, but achieve so in mode 3. This is coherent with the initial hypothesis of this work, which was that better results can be achieved with the extension of Apollo, since the system can now choose the transformation that best fits the kernel under study from a finite set of transformations. Additionally, this also shows that, as expected, better results can be expected when increasing the timeout granted to Pluto to find a suitable transformation, although this usually comes at the cost of more iterations needed to cancel out this overhead. This specific hypothesis will be assessed later in this study.

However, when comparing the results obtained by the multiversioning system with those obtained by Clang, the possible outcomes vary considerably. As expected, the multiversioning system is able to outperform Clang results for the largest datasets, but fails to do so with the smallest ones, especially in PolyBench. Specifically, as shown in Table 8-11 and Table 8-12, regardless of the timeout, the multiversioning system is not able to outperform Clang in almost any of the `Small` or `Medium` problem sizes. This is coherent with the conclusions extracted in the study of the transformation selection: for the smallest problem sizes, the multiversioning system almost never chooses a transformation other than the original code. In turn, this justifies the negative speedup achieved in those cases; as already explained, executing the original version of the code through Apollo implies a non-negligible overhead for the smallest problem sizes. Consequently, it is expected that Clang version will potentially outperform the multiversioning system every time the original code is chosen. However, this still implies an important improvement from the point of view of Apollo: since Apollo did not consider the possibility of the original code being better than the transformed one, if a transformation is successfully generated, it is maintained until the end, even if the resulting code is much worse performing than the original one. With the possibility of choosing the original code as the best performing, this negative impact is considerably reduced; this is one of the reasons why the multiversioning system presents rather large speedups with respect to standard Apollo in the smaller problem sizes, especially in the `Small` category.

Additionally, it can be seen that, in general lines, the number of times the multiversioning system fails to outperform Clang is smaller when using an infinite timeout than when using the $2\times$ timeout. For instance, with the $2\times$ timeout, 53 cases fail both in Namaka –8 from ApolloBench and 45 from PolyBench–, and in Magerit –6 from ApolloBench and 47 from PolyBench–. On the contrary, when using mode 3, these

numbers are reduced to 44 in Namaka –4 from ApolloBench, 40 from PolyBench– and to 41 in Magerit –5 from ApolloBench, 36 from PolyBench–. Nonetheless, it should be noted that the great majority of them belong to the smallest problem sizes, which also justifies why the difference between both modes is rather small.

To conclude the study of the cases in which the multiversioning system fails to outperform Clang results, there are three cases that entail a special relevance: `cholesky`, `trisolv` and `jacobi-1d`, which have been highlighted in blue in Table 8-11 and Table 8-12. These cases were already pointed out in Subsubsection 8.2.1.2, since, regardless of the working mode, the multiversioning system failed to find a transformation better performing than the original code. In the mentioned section, three possible causes for this behavior were considered: (i) that the timeout was not large enough for Pluto to generate any transformation; (ii) that, even for the largest problem sizes, the workload was still too low for any optimization to overcome the original code; or (iii) that the set of transformations chosen and presented in Table 8-2 was not the most suitable one for these specific tests. The first reason was directly discarded in the mentioned study, since this behavior remained when using the infinite timeout, but to choose between the other two, the performance analysis was necessary. Now, with the numbers provided in Table 8-7, Table 8-11 and Table 8-12, this study can be finished. To evaluate if there is enough evidence to support the second reason, Table 8-7 must be analyzed again. As can be observed, with respect to the three tests mentioned, both `trisolv` and `jacobi-1d` fulfill this criterion, since, even for the Extralarge category, their execution time is just in the order of a few milliseconds. This is coherent with the results obtained in both the transformation selection and the performance studies: in those cases, no transformation was ever selected and, consequently, no significant speedup is ever achieved for those tests, regardless of the target architecture. However, this does not hold for `cholesky`: although, as in all the Small and Medium cases, its workload is too small for the multiversioning system to provide efficient results, the Large and Extralarge categories present sequential times that cannot be considered negligible anymore. Consequently, the conclusion that can be extracted is that the transformations tested are not suitable for this specific test, since they are tried but never selected and, thus, no speedup is ever achieved.

Furthermore, one of the most relevant conclusions that can be extracted from these tables is related to the overhead added by the training phase. As can be observed, regardless of the timeout, the average training time per invocation is, in general lines, considerably larger than the operational time. This is due to two main reasons: first, one of the expected outcomes of the multiversioning system is a speedup due to memoization, since some time-consuming stages of Apollo, such as the instrumentation and the transformation selection –in which Pluto is invoked– are directly skipped, so a performance gain is expected from this feature when comparing the training phase and the operational phase; secondly, as already mentioned, experimenting with different transformations can lead to a significant performance improvement, but can also result in rather worse execution times: for instance, although tiling in the L2 cache –by activating `l2tile`– can be beneficial for

some specific codes, its usage in other codes can lead to very poor performances. The effect of memoization can be directly noticed by observing the speedups with respect to standard Apollo: when this number is very close to 1, a plausible explanation is that the only improvement with respect to the transformation tested by standard Apollo comes from skipping some time-consuming stages, as the call to Pluto. Several examples of this behavior –highlighted in yellow– can be found in Table 8-11 and Table 8-12, such as `floyd-warshall` and `jacobi-2d` in their Large and Extralarge versions, or `fdtd-2d` and `heat-3d` in their Extralarge version.

On the other hand, the effect of testing inefficient transformations can be clearly identified in the tables too, since there are some cases in which the average training time is excessively large if compared to the operational time. This feature has also a direct impact in the number of invocations needed to overcome either of the baselines: testing inefficient transformations, together with rather small speedups, makes these figures of merit grow considerably. In those cases, the overhead induced by the training phase is usually too large to compensate the usage of such a system either with respect to Clang or to standard Apollo. To exemplify this behavior, two cases, `needle-Small` and `trmm-Medium`, have been highlighted in red in Table 8-10 and Table 8-12. As can be observed, these two cases present the two issues aforementioned when comparing the multiversioning system with the results obtained by Clang: first, the average training time is excessively large when compared to the operational time –0.772s versus 0.37ms for `needle-Small`, and 1.569s vs 5ms for `trmm-Medium`–; secondly, the speedups achieved are rather small –1.129 $\times$ for `needle-Small` and 1.046 $\times$ for `trmm-Medium`–. As expected, the combination of these two issues results in an increment in the number of invocations needed to overcome, in this case, Clang: in both cases, this number is greater than 100,000, and way above the rest of the experiments.

To conclude this analysis, one of the cases is going to be studied in depth. The case selected is `pcg`, and it has been chosen because it exemplifies many of the behaviors mentioned in this study, thus being a good example to highlight the advantages and disadvantages of such a system. First, `pcg-Small` executed in Namaka is going to be thoroughly evaluated with the objective of comparing the differences between training and operational phases in the two working modes; afterwards, the three different problem sizes of `pcg`, this time executed in Magerit, are going to be used to illustrate how the system adapts to different problem sizes.

Figure 8-1 represents the evolution of the execution times measured in Namaka for `pcg-Small` when the multiversioning system is working. The blue series represents the execution times measured for this kernel with a 2 $\times$ timeout, while the orange series gathers the same information, but with an infinite timeout. Each pair of bars represents the measured execution time for the invocation number specified by the x axis with each working mode: the first pair represents the execution time measured for invocation 0, the next pair represents invocation 1, and so on. Consequently, the first 15 pairs represent the execution times measured for each of the invocations composing the training phase; likewise, the rest of the pairs represent the execution times measured for the first 5 invocations of the operational phase. Additionally,

as already mentioned, each iteration of the training phase is devoted to test one of the transformations proposed in the transformation set gathered in Table 8-2; therefore, invocation 0 evaluates transformation 0 –the original code–, invocation 1 assesses transformation 1, and so on, until completing the set after invocation 14, thus triggering the operational phase in successive invocations.

Figure 8-1: Overview of the behavior of pcg-Small kernel during training and operational phases of the multiversioning system.

From this figure, many interesting conclusions can be extracted:

- One of the main features drawing the attention is the difference in the execution times of the training phase when comparing both timeouts. As can be observed, the times measured with the infinite timeout are considerably larger than those measured with the $2\times$ timeout. This is coherent with all initial hypotheses: in the infinite timeout, the amount of time that Apollo can invest in generating the transformation is not limited; hence, if there are transformations requiring longer times, a performance loss in those invocations is expected.
- Although the trend is that the training takes longer with the infinite timeout, there is a case that stands out, namely the invocation in which transformation 14 is being tested. In this case, the trend is maintained, but it can be seen that the execution time for the infinite timeout is excessively large: the execution time for the infinite case is 34.4 seconds, but less than 4 seconds for the $2\times$ timeout. This excessive time can be due to two main reasons: first, that the transformation takes a long time to be generated; secondly, that the transformation is inefficient, and thus, the performance worsens, instead of improving. Considering that the transformation tested is the one trying to optimize the L2 cache usage, both reasons are plausible: Pluto takes longer in its generation and, on top of that, this transformation may be very beneficial for some codes –as needle-Large in Namaka, in which it is selected as the best performing one–, but rather inefficient in others, such as this one. To avoid these situations, when executing

the optimized code, another timeout could be added in such a way that, if the transformation being executed is taking too long, it is aborted and automatically discarded, so as to not add this large overhead. However, to do so, the static component of Apollo needs to be deeply modified; since this is out of the scope of this work, it is proposed as a future line in Section 9.4.

- Regarding the operational phase, it can be observed that the performance achieved with the infinite timeout is larger than that obtained with the $2\times$ one. This is also coherent with all the initial hypotheses: granting more time to Apollo can help in improving the ideal performance achievable, since there can be some transformations that, although useful, need more time to be generated; hence, with strict timeouts, these transformations may not be evaluated. This specific case exemplifies this behavior: with the small timeout, the transformation selected is transformation 0 –i.e., the original version and highlighted with a blue arrow in Figure 8-1–; with the large timeout, the transformation selected is now transformation 9 –highlighted with an orange arrow–. Regarding the number of transformations tested with each timeout, only 3 can be tested with the strict timeout, while all of them are evaluated with the large one. Consequently, this case also serves to show the importance of looking for a suitable value for the timeout: as shown in Table 8-9, with the strict timeout the multiversioning system is not able to outperform Clang; on the contrary, Table 8-10 shows that the largest timeout succeeds in doing so, but at the cost of needing 3424 invocations to start canceling out the added overhead.
- Finally, to conclude this analysis, a clarification about the timeouts must be made. As described at the beginning of this chapter, a timeout of $2\times$ means that, to generate a valid transformation, Apollo grants the transformation selection stage a time equal to twice the time taken by the sequential thread executing the original version of the code. If this timeout expires, then the transformation selection is aborted and the results of the sequential thread are retrieved. This timeout is conceived as a mechanism to extend the transformation selection process while limiting the overhead considered acceptable; nonetheless, defining a timeout of $2\times$ does not mean that the code will take up to twice the original code to execute. As already described at the beginning of this section, this timeout has been implemented as a series of checkpoints, not as interruptions: the transformation selection stage is composed of several sequential operations, and the verification of the timeout is performed before and after each of these operations; if the timeout expires during one of these operations, the process is not stopped until this operation finishes. Hence, if Pluto has already been called, it will finish nonetheless. For that reason, it should be clearly stated that the timeout is tentative, not strict.

Finally, the results obtained with pcg for the 3 problem sizes and the 2 working modes in Magerit are also going to be analyzed. This kernel has been chosen because it summarizes many of the possible outcomes foreseen when devising the

multiversioning system, so it is a good candidate to sum up the functional validation from the polyhedral point of view. To this end, Table 8-13 repeats the information needed to ease this analysis, including the baseline times and the transformation selected –denoted as *Clang*, *ApolloS* and *Tx*, respectively.

Table 8-13: Performance results obtained for pcg kernel in Magerit (in seconds).

Size	Clang	ApolloS	Mode	Train	Oper	SpdpC	SpdpS	InvC	InvS	Tx
S	1.136	1.893	2×	4.483	1.988	0.571	0.952	–	–	1
			infinite	11.985	0.277	4.092	6.823	205	109	7
M	3.912	2.243	2×	9.959	0.385	10.167	5.829	41	78	3
			infinite	13.651	0.425	9.213	5.282	57	110	7
L	40.083	7.242	2×	33.645	2.355	17.023	3.076	1	97	7
			infinite	34.196	2.331	17.194	3.107	1	98	7

One of the results that draws attention is that obtained with the 2× timeout for the Small problem size. As can be seen, this result seems weird, since the operational phase takes almost as much time as for the Large category. What is happening here is that, for this specific case, the multiversioning system is only able to evaluate the original code and transformation 1, which was selected precisely because it is the one requiring less time to be generated. However, the performance of this transformation is very similar to the performance of the original code, and hence no speedup is achieved. Nonetheless, this behavior changes completely when increasing the time allowed to Pluto to generate the transformation: as can be seen, with the infinite timeout the operational phase reaches a considerable speedup, reducing the execution time from 1.988 seconds to 0.277 seconds thanks to transformation 7, but at the cost of requiring 205 and 109 invocations to overcome Clang and standard Apollo, respectively.

With respect to Medium, the results obtained are also interesting, since they show the behavior of the multiversioning system when two transformations yield similar results. As can be observed, transformation 3 is chosen for the small timeout, while transformation 7 is chosen for the large one. This example represents a case in which the 2× timeout is actually more efficient than the infinite one, since the performance achieved is similar, and the overhead added in the training phase is significantly smaller, thus reducing the number of invocations needed to overcome either baseline.

Finally, in the case of the Large category, it can be observed that both timeouts yield the same result, since the same transformation is always chosen and the performances and overheads are also similar.

To sum up, this section has presented the performance analysis of the benchmarks selected to evaluate the multiversioning system proposed in this document, which, together with the transformation selection study carried out in Subsubsection 8.2.1.2, completes the functional validation from the polyhedral domain point of view. The main conclusions extracted from this study are:

- Regarding the comparison with the original version of Apollo, it has been proven that the multiversioning system always provides better results when granted

enough time, proving correct the initial hypothesis that such a system can improve considerably the performance achieved by applications presenting a dataflow-like behavior, in which the same kernel is invoked many times.

- It has also been shown that, as expected, this kind of system provides better results for larger problem sizes. For smaller problem sizes, it has also been proven to provide very good results, but usually at the cost of adding an overhead that, depending on the application, may be considered too large.
- Consequently, it has also been proven that the timeout must be carefully selected, looking for a tradeoff between the speedup achieved and the overhead introduced to that end.
- Finally, it has been observed that, as expected, the transformation selected has an important impact in the performance achieved, and that their efficiency varies depending on the kernel, the problem size, the timeout and the target architecture. This proves correct the main idea that motivated the extension of Apollo: devising a system that automatically specializes the optimizations applied to a code depending on its environment conditions can have a great impact on the performance achieved, extending the state of the art related to automatic acceleration of dataflow-based applications.

This concludes the functional validation of the proposed methodology from the polyhedral point of view. The remaining of this section is going to also address the functional validation, but this time from the dataflow domain.

8.2.2 Dataflow benchmarks

Once the methodology has been functionally validated from the polyhedral domain point of view, the same must be carried out for the dataflow domain. To do so, three different dataflow applications are going to be used: Sobel-morpho, Stereo matching and Matrix multiplication.

The objective of the previous section was to prove that the extensions and modifications addressed within Apollo work as expected and provide the foreseen benefits, which were the reason why these changes were introduced in the first place. Nonetheless, the objective of this section is slightly different, since its main goal is to prove that extending a dataflow framework as PREESM –as well as its dynamic counterpart, SPiDER– through automatic optimization techniques, such as those provided by Apollo, results in a positive impact in the performance achieved.

As described in Chapter 6, combining a dataflow framework such as PREESM –and SPiDER– with a polyhedral optimizer like Apollo involves balancing the parallelization possibilities allowed to each one. Hence, one of the objectives of this section is (i) to illustrate how this balancing can be addressed and (ii) to analyze if the foreseen benefits are actually appearing or not. To do so, this subsection is organized as follows:

- First, the dataflow implementations of the three applications used in this section are going to be described, together with the different problem sizes that are going to be configured for each one, similarly to what was done in the previous section.
- Afterwards, the baseline implementations are going to be characterized in terms of execution time, considering as baselines the implementations that do not include any type of parallelism –neither in PREESM nor in Apollo.
- Once these two steps have been fulfilled, the next experiments will be focused on assessing the performance achieved when playing with the different parallelization possibilities. To this end, three different situations are devised:
 - In the first one, all the parallelization possibilities are granted to PREESM. This is the case already existing in the literature for dataflow applications, in which their structural parallelism can be exploited, but without providing them with the possibilities offered by automatic optimizers, such as Apollo.
 - The second one represents the opposite case, namely letting all the parallelization possibilities to Apollo. In this case, the potential of Apollo as an automatic parallelizer is fully exploited, but the exploitation of the structural parallelism –provided by PREESM– remains unexplored.
 - The third case represents the combination of both tools: all the parallelization possibilities are granted to PREESM, but Apollo is also used, although just as an optimizer. This case allows the exploitation of both types of parallelism: while PREESM exploits the structural parallelism, Apollo tries to optimize important features, such as the memory usage, so as to speed up the execution.
- Finally, this section is going to be closed with the functional validation of SPiDER, the dynamic scheduler of PREESM. Up to this point, the only feature still requiring a validation is the existence of dynamic parameters. These dynamic parameters may affect the structure of the application and, hence, the implementation itself, so they must be taken into consideration by the polyhedral optimizer, as described in Section 5.5. Therefore, this last study will focus on the behavior of the proposed system when these parameters appear.

Consequently, the remainder of this section is going to be organized as follows: first, Subsubsection 8.2.2.1 will present the dataflow implementations of the applications, together with the problem sizes that are going to be used; afterwards, Subsubsection 8.2.2.2 will characterize the different throughputs achieved when balancing the different parallelization possibilities offered by Apollo and PREESM; finally, Subsubsection 8.2.2.3 will present the validation of the methodology in a dynamic environment, with parameters changing at runtime.

8.2.2.1 Specific setup

As aforementioned, before getting to the performance study, the dataflow implementations of the applications selected for this analysis are going to be described, as well as the problem sizes that have been chosen for each one. As already established, three different applications are going to be used: Sobel-morpho, Stereo matching and Matrix multiplication, which will be hereafter referred to as Sobel, Stereo and Matmul. Although the functionality of these applications was already described in Subsection 7.1.1, a brief summary is going to be provided here, together with a detailed description of the dataflow implementation.

Sobel application contains a filter widely used in image processing, usually as part of edge detection algorithms. The specific dataflow implementation in PREESM is shown in Figure 8-2, which consists of 6 actors and 9 parameters. Regarding the actors, *Read_YUV* and *display* are devoted to reading the input file and displaying the output, *Split* is responsible for splitting the input sequence so it can be processed in parallel, *Sobel* contains the implementation of the operator and *Dilation* and *Erosion* are also known techniques in image processing. Regarding the parameters, it can be seen that 5 of them need to be specifically defined by the developer, while the other 4 depend on those ones. The 5 independent parameters are *width* and *height*, which correspond with the size of the input sequence; *nbSlice*, which represents the number of slices in which the input is going to be divided to be processed in parallel; *window*, which is an input of *Erosion* and *Dilation* actors; and *index*, which is an identifier for the display. The rest of the parameters are related to adapting the application so it can be correctly parallelized by processing several slices of the input simultaneously. Consequently, this implementation has been manually adapted so PREESM can exploit the structural parallelism: by modifying the value of *nbSlice*, the network is automatically reconfigured to generate as many slices of the image as that specific value indicates; then, in the code generation stage of PREESM, users can select how many processing cores they want to put in use, and the code is generated automatically and distributed among those cores.

Figure 8-2: PREESM application graph of Sobel filter.

Stereo algorithm aims at processing two images coming from different sensors to generate a disparity map that expresses the depth of the captured scene, being one of the most common techniques to determine 3D depth information. The top-level graph of the PREESM dataflow implementation can be found in Figure 8-3. As can be observed, the network is quite complex, including many actors and parameters. However, they can be easily grouped regarding their functionality:

- Regarding the actors, *Cost_Parallel_Work* is a hierarchical actor containing all the actors involved in the computation of the matching cost of each pixel for a given disparity and the subsequent production of a disparity map, as shown in Figure 8-4; *Median_Filter* applies a median filter to the input disparity map with the objective of smoothing the results; as for the rest of the actors, they are related to reading the inputs, writing the outputs, or adapting the network so PREESM can successfully exploit its structural parallelism.
- Regarding the parameters, *height* and *width* represent again the dimensions of the inputs; *nbSlices* denotes the pieces in which the input images can be divided to be processed simultaneously; *minDisparity* and *maxDisparity* represent the minimum and maximum values that can be found in the output disparity map, respectively; *nbIterations* indicates how many times the cost must be computed; and the rest of them are also devoted to adjust the application to be parallelized.

Matmul application computes a matrix-matrix multiplication ($C = A \times B$), and the top-level graph of its dataflow implementation in PREESM is provided in Figure 8-5. As can be observed, it is composed of 5 actors and 5 parameters: regarding the actors, *matrixGen_0* and *matrixGen_1* are devoted to generating the input matrices, *matrixMult* computes the multiplication, *writer* stores the resulting matrix, and *BroadcastMatrixB* is a special actor for easing the parallelization process by broadcasting matrix *B* to all the instances of *matrixMult* actor; regarding the parameters, *rowsA*, *colsA* and *colsB* represent the dimensions of *A* and *B*—it should be noted that there is no *rowsB* because it has to be equal to *colsA* for the multiplication to be feasible—, *Parallelism* denotes the degree of parallelism the user wishes to provide the algorithm with, and *rowsParallelized* is inferred from the other parameters and is necessary for easing the parallelization in such a way that it represents how many rows are going to be processed by each instance of *MatrixMult* actor.

Before getting into the analysis of the experiments, the problem sizes must be defined. As shown in Table 8-14, two different problem sizes for the input data have been defined, *Small* and *Large*. In general lines, these problem sizes have been selected so different behaviors can be observed: with the *Small* size, it is expected that the workload is not large enough for the combination of tools to provide considerable speedups; in that case, the most possible outcome is that the best throughputs are achieved with either PREESM or Apollo separately; however, with the *Large* size, the workload is expected to be large enough to be a representative example of the cases in which the combination of both tools results in a positive impact in the throughput.

Figure 8-3: PRESM application graph of Stereo.

Figure 8-4: PREESM application graph of the *Cost_Parallel_Work* actor within Stereo.

Figure 8-5: PREESM application graph of Matmul.

Table 8-14: Summary of the different problem sizes for each dataflow application.

Application	Small	Large
Sobel	1800×1500	3840×2160
Stereo	450×375	1800×1500
Matmul	320×320	3200×3200

Once the implementations and the problem sizes have been reviewed, the rest of the section will assess the performance achieved when using both PREESM and Apollo, as well as the reactivity of the proposed methodology to changes at runtime through SPiDER. Since the objective of these studies is to evaluate the impact this combination of tools may have in the throughput, the timeout configured for Apollo is the infinite one, since it is expected to be the one providing the largest speedup. To configure this timeout within PREESM, the working mode selected for Apollo is mode 4; as described in Chapter 6, the main characteristics of this mode are that it establishes an absolute, configurable timeout, and that the sequential thread launched within Apollo to mask Pluto's overhead is not launched anymore so as to not interfere with PREESM's thread management. Consequently, this mode has been configured with such a large value for the timeout that it can be considered as infinite.

8.2.2.2 Static performance evaluation

As mentioned before, the objective of the study presented in this section is to compare the throughput that can be achieved with the three applications when using PREESM and Apollo separately, and when combining them. As happened with the functional validation of the polyhedral extensions, this study will be conducted both in Magerit and Namaka and, to that end, the following experiments have been carried out:

- First, the sequential versions –considered as the baseline– of all the tests and problem sizes have been evaluated. These sequential versions are created when deactivating the parallelization both in PREESM and Apollo and generating the code for 1 core using PREESM. Hereafter, this version will be known as *Seq*. It should be noted that, since the baselines of the study conducted in Subsubsection 8.2.1.3 were generated with Clang, for homogeneity reasons the default compiler for these baselines will be also Clang, instead of gcc.
- The next versions to be evaluated are those in which the parallelization potential of each tool is assessed separately: first, when exploiting the structural parallelism through PREESM; afterwards, when fully exploiting the parallelization potential of Apollo, in detriment of PREESM. Hereafter, these versions are going to be known as *PREESMPar* and *ApolloPar*, respectively.
- Finally, the last version to be evaluated is the one conjugating both parallelization potentials: PREESM exploits the structural parallelism and Apollo further optimizes the code. Hereafter, this version is going to be known as *CombPar*.

It should be noted that the level of parallelism differs from one architecture to another: as established in Section 8.1, Namaka gathers 16 processing cores, while the node that is being used in Magerit supports up to 20 threads. Hence, those are the levels of parallelism defined in PREESM for the implementations in which PREESM tries to exploit the structural parallelism. Additionally, the profiling of the actors within each application will be performed by means of PAPIFY, which is a monitoring toolbox that simplifies the profiling by retrieving hardware information from PMCs and which is automatically included within PREESM framework, as described in Section 4.2.

Table 8-15 gathers the results obtained for the aforementioned applications in Namaka and Magerit. As already said, dataflow applications are executed in a loop; hence, the times presented in Table 8-15 define the average execution time per iteration of the dataflow graph. These results are organized as follows: for each application and problem size, first the times measured for the sequential version and the one in which PREESM parallelizes are presented; afterwards, the execution times for the two types of experiments in which Apollo is involved –*ApolloPar* and *CombPar*– are provided. These latter results are, in turn, divided into several categories, as in the functional validation addressed in Subsection 8.2.1: *Train* provides the average time spent per invocation in evaluating the 15 transformations of the training set; *Oper*

provides the average time per invocation once the system goes into the operational phase; *InvC* represents the number of invocations required by the multiversioning system to start providing speedups with respect to the corresponding baseline –i.e., the sequential version for *ApolloPar*, and *PREESMPar* for *CombPar*–, or a dash if no speedup is ever achieved; and *Tx* provides the identifier of the transformation that has been selected. Regarding the latter, it should be noted that, for the *CombPar* experiment, a notation different from the one shown in Table 8-2 is going to be used. The rationale behind this change is that, in this experiment, Apollo is used just as an optimizer, so the `-parallel` flag is deactivated for all the transformations –refer to Table 5-1 for more information about the flags–. Consequently, to highlight this change, the transformation identifier is going to be the same than the one presented in Table 8-2, but starting with a 0. The analysis of Table 8-15 brings out some interesting features that are going to be detailed below.

Table 8-15: Performance results obtained for the dataflow applications (in seconds).

Program			Seq	PREESM Par	ApolloPar				CombPar			
					Train	Oper	InvC	Tx	Train	Oper	InvC	Tx
Sobel	S	Namaka	0.093	0.012	0.225	0.093	–	12	0.906	0.012	–	02
		Magerit	0.110	0.029	0.323	0.112	–	2	1.221	0.028	–	08
	L	Namaka	0.374	0.037	0.512	0.374	–	6	1.615	0.039	–	02
		Magerit	0.439	0.082	0.658	0.440	–	2	1.742	0.083	–	05
Stereo	S	Namaka	0.313	0.214	4.769	0.312	–	1	1.937	0.215	–	00
		Magerit	0.674	0.308	8.791	0.646	4363	2	1.392	0.308	–	00
	L	Namaka	5.169	3.927	10.045	5.167	–	1	4.362	3.927	–	00
		Magerit	7.021	5.135	15.341	7.022	–	2	4.104	5.136	–	00
Matmul	S	Namaka	0.026	0.002	0.229	0.002	146	2	1.692	0.001	28175	02
		Magerit	0.041	0.003	0.344	0.002	134	12	1.547	0.001	17832	02
	L	Namaka	131.091	10.443	12.138	1.711	1	12	7.549	1.359	1	09
		Magerit	142.074	13.141	12.670	2.165	1	2	8.979	1.102	1	09

First of all, as it can be observed, Matmul is the application achieving the best performance results, since the experiments where Apollo is being used prove to be the cases in which fastest execution times are achieved. As shown in Table 8-15, *CombPar* is the case in which the best throughput is achieved, regardless of the target architecture and problem size. Regarding the latter, it can be seen that Apollo succeeds in accelerating the application even for the Small problem size, whose computational workload is rather short. For that reason, the number of invocations needed to overcome the baselines is quite large, especially in the *CombPar* case; as PREESMPar already provides a considerable speedup, canceling out the overhead added by the training phase requires a larger amount of invocations. In conclusion, the behavior of Matmul application is coherent with the initial hypothesis, proving that the combination of both tools –and the subsequent extension of Apollo to comply with dataflow applications and further exploit the possibilities they offer– can lead to better performing results than those obtained separately.

On the contrary, it can be seen that, in general, there is no apparent difference between the results obtained for Sobel and Stereo with PREESM and with Apollo, since the execution times measured are similar when comparing each version of Apollo with its corresponding baseline. In the case of Stereo-*CombPar*, this behavior is easily justified, since, as shown in Table 8-15, Apollo chooses the original transformation regardless of size and architecture. However, for the rest of the cases, a transformation is chosen, but no speedup can be appreciated from the results gathered in this table.

To understand what is happening, a deep profiling is in order, detailing not only the execution time per iteration of the dataflow graph, but also the timings associated to the actors containing kernels targeted by Apollo. As shown in Figure 8-2, Figure 8-3, Figure 8-4 and Figure 8-5, the number of actors conforming each application varies from one to another, being Matmul the one with fewer actors and Stereo the one with more. Hence, the first step towards clarifying the obtained results is to analyze (i) the workload distribution of each application in the sequential versions, so as to highlight the actors requiring more execution time; and (ii) the actors that are being targeted by Apollo. As the workload distribution is going to be analyzed for the sequential version, performing this study for both architectures is somehow redundant, so Table 8-16 gathers the results for just one architecture, namely Namaka. From the observation of this table, the justification of some of the behaviors detected in Table 8-15 comes to light, since the actors in which Apollo is applied have very different workloads:

- With respect to Sobel application, the only actor Apollo targets is actor *Sobel*, which supposes barely a 2% of the total workload, justifying why no apparent speedup is observed in the average time per iteration of the application graph. As can be seen, *Erosion* and *Dilation* actors accumulate a 97% of the workload; however, Apollo is not successful in optimizing them, since no prediction model can be built. Hence, as only a 2% of the execution time is targeted, to observe the speedup one must analyze the specific times of the actor being optimized.
- As happened with Sobel, in the case of Stereo application, only one actor is being targeted by Apollo, and this actor represents less than a 20% of the total execution time. For the rest of the actors accumulating most of the workload, Apollo again cannot build the prediction model. Hence, this can also be the reason why there is no apparent speedup in this case when Apollo is used.
- On the contrary, regarding Matmul application, Apollo is able to optimize *matrixMult* actor, which gathers a 97% of the total execution time. Since the other actors can be considered negligible, Apollo is not applied. Consequently, it seems evident that this is the main reason why the speedup is clearly observed in the average time per iteration of the dataflow application graph.

From the analysis of Table 8-16, one can conclude that, to observe the effect of Apollo in these applications, the execution time of the targeted actors must be studied. These times are gathered in Table 8-17; it should be noted that the results obtained

Table 8-16: Workload distribution analysis of the dataflow benchmarks in Namaka (in %).

Application	Actor	Small	Large	Apollo?
Sobel	<i>Erosion</i>	48%	48%	✗
	<i>Dilation</i>	49%	49%	✗
	<i>Sobel</i>	2%	2%	✓
	<i>Others</i>	1%	1%	✗
Stereo	<i>AggregateCost</i>	56%	56%	✗
	<i>ComputeWeights</i>	19%	18%	✓
	<i>CostConstruction</i>	15%	13%	✗
	<i>MedianFilter</i>	4%	5%	✗
	<i>Others</i>	6%	8%	✗
Matmul	<i>matrixMult</i>	97%	97%	✓
	<i>matrixGen</i>	2%	2%	✗
	<i>writer</i>	1%	1%	✗

for Matmul application do not appear in this table, since *matrixMult* actor consumes a 97% of the total execution time in Namaka and, therefore, this study is not necessary for *Matmul* application. On the contrary, for Sobel and Stereo applications, the information provided in this table sheds some light to understand what is happening.

With respect to Sobel, it can be seen that *ApolloPar* configuration succeeds in overcoming the performance of the sequential version, while *CombPar* fails to overcome *PREESMPar*. This proves that, for this application, exploiting the structural parallelism by means of *PREESM* results in better performance than that provided by Apollo, since the best performance is achieved with the *PREESMPar* configuration.

Regarding Stereo, as aforementioned, *CombPar* fails to optimize the code, since the transformation selected is always the original code and, thus, the performance achieved in this case is similar to that obtained with *PREESMPar*. On the contrary, it can be noticed that *ApolloPar* is successful in improving the throughput, overcoming not only the sequential version, but also *PREESMPar*, resulting in the best performing configuration. From these results, it can be inferred that, for this application, Apollo provides satisfactory results when it is in charge of the parallelization, but fails to provide any further improvement when used just as an optimizer. Regarding *PREESMPar*, it should be noted that the reason why it fails to exploit the parallelization of this specific actor is because the parameter defining the level of parallelism, *nbSlices*, does not affect this actor; consequently, this specific actor is not individually targeted by *PREESM*. The only aspect in which *PREESM* can improve the timings is when scheduling several instances of this actor to be executed simultaneously in different cores, but the actor itself is not divided. Hence, this is a good case to show that Apollo can improve the performance beyond the possibilities offered by *PREESM*.

In both cases, a detail drawing the attention is the difference in the training times between *ApolloPar* and *CombPar*. As can be seen, regardless of size and platform, the training time in the *CombPar* configuration is much larger than in *ApolloPar*. The

rationale behind this is that, although both Apollo and all the tools conforming it have been adapted to be used in a multithreaded environment, there are still some parts that must be executed in mutual exclusion to ensure no data races ever occur. This implies that the larger the number of threads trying to access the same resource at the same time, the larger the waiting until they acquire the mentioned resource. In *ApolloPar*, only one call to Apollo is being performed simultaneously, while in *CombPar*, several threads –16 in Namaka and 20 in Magerit– are invoking Apollo at the same time. Hence, *CombPar* adds a non-negligible overhead in multithreaded contexts, which is bound to worsen when the number of threads accessing Apollo at the same time increases, but which disappears once the training phase is over.

Table 8-17: Detail of the performance results obtained for the actors in which Apollo is invoked in Namaka (in milliseconds).

Program		Seq	PREESM Par	ApolloPar				CombPar			
				Train	Oper	InvC	Tx	Train	Oper	InvC	Tx
Sobel	S	1.884	0.213	32.113	0.800	434	12	517.895	0.442	–	02
	L	7.213	0.797	58.458	1.944	161	6	875.501	1.306	–	02
Stereo	S	5.721	5.714	382.196	1.343	1305	1	842.997	5.713	–	00
	L	91.869	91.659	482.413	19.113	96	1	936.542	94.019	–	00

To close this study, Figure 8-6 details the speedups achieved in Namaka for the actors targeted by Apollo. The bars denote the speedup of each experiment when compared to the sequential version. This figure summarizes the main conclusions extracted from the functional validation through dataflow benchmarks:

- For Sobel, it shows that the best performance is reached without Apollo, when only PREESM is exploiting the parallelism. With respect to Apollo, it can be seen that some speedups appear when comparing with the sequential version –around 4× for *ApolloPar* and 6× for *CombPar*–, but they are always below the one provided by PREESM –around 9×–. As can be observed, similar performance results are obtained regardless of the problem size.
- As for Stereo, the best performance is achieved by *ApolloPar* –around 5×–, being a good example of a case in which Apollo is able to optimize an actor and PREESM fails to do so. The other two cases do not provide any speedup, as PREESM does not parallelize the targeted actor and Apollo does not provide any improvement when the `-parallel` flag is deactivated.
- Matmul application is, without any doubt, the one exemplifying the best the advantages the combination of both tools can offer. As can be seen, the speedups reached when using Apollo are always larger than those obtained by PREESM, even though PREESM succeeds in parallelizing the application –for both problem sizes, the speedups are around 14× in a 16-core architecture–. Specifically, with the *CombPar* configuration, the speedup achieved is much larger than the one expected when using the 16 cores of the target architecture: 25× for the Small

problem size, and almost $100\times$ for the Large one. This huge difference is due to the many optimizations a tool like Apollo is able to perform, apart from parallelizing the code. These optimizations are usually related to improving data locality, hence contributing to making an efficient memory usage.

Figure 8-6: Speedups achieved with *PREESMPar*, *ApolloPar* and *CombPar* for the actors targeted by Apollo in Namaka.

As aforementioned, one of the aspects standing out the most is the surprisingly large value of the speedups achieved by Apollo in Matmul application. The explanation given to this behavior is that Apollo does not only parallelize, but also applies some optimizing transformations devoted to making an efficient memory usage. In other words, there may be several sources from which a speedup may appear: the most obvious one is the increment of the number of PEs in such a way that, with a 16-core architecture, an ideal speedup of $16\times$ is expected; however, there may be other sources, such as a more efficient usage of memory or, for example, an increment in the frequency at which the PEs are running. These two examples, in fact, can cause speedups to appear even in 1-core scenarios, that is, when no parallelization has been addressed and, thus, no speedup –of the first type described above– can be expected.

To further illustrate this behavior, a brief example is going to be presented, using to that end Matmul application with the Large problem size. In this experiment, PAPIFY tool has been used not only to monitor the execution time, but also to access some PMCs to retrieve the information of two hardware events: L1-Instruction Cache Misses (ICM) and L1-Data Cache Misses (DCM). Through this tool, the number of occurrences of these events during the execution of each actor can be counted, so it can be of great help to illustrate how Apollo is using the cache memory. The objective of this experiment is thus to compare the cache memory usage of PREESM –using Clang– and Apollo, specifically when the latter is acting not as a parallelizer, but as an optimizer. Hence, for this experiment, Matmul is executed first with Clang –i.e., the

sequential version described before– and afterwards with *ApolloPar*, but deactivating the `-parallel` flag to ensure that Apollo is acting as an optimizer –i.e., the same code than with Clang, but with Apollo optimizing the memory usage–. Figure 8-7 gathers the results retrieved by PAPIFY for L1-ICM and L1-DCM events in both cases, together with the execution time associated to each case. As can be seen, the difference in the number of occurrences of both events is more than one order of magnitude between Clang and Apollo, and the difference in the execution times is similar, since a speedup of $7\times$ is observed. This justifies why the speedups achieved surpass by far the $16\times$ expected from a 16-core architecture: even in a 1-core scenario like the one presented here, there is a speedup, despite the number of processors not being increased; as there are considerably less cache misses, the processor reduces the time it is stalled, thus increasing the throughput. This experiment is also useful for showing the advantages of combining the parallelization potentials: when combining the speedups achieved by PREESM by itself –around $14\times$ – with the one offered by the optimizations applied through Apollo – $7\times$ –, a speedup near $100\times$ is reached.

Figure 8-7: Detail of the performance indicators measured with PAPIFY (L1-ICM, L1-DCM and execution time) for Matmul-Large in Namaka with Clang and Apollo.

The studies presented in this section have shown the advantages and disadvantages of combining a dataflow tool like PREESM with a polyhedral framework like Apollo when targeting traditional dataflow applications. The three applications selected for this study have proven to be a good set to perform this functional validation, since they provide representative examples of applications where granting the parallelization potential to one tool or the other results in very different performance results: Sobel achieved better results when PREESM was in charge of parallelizing, Stereo reached larger speedups when Apollo was in charge, and Matmul proved to be much faster when the optimizations came from both frameworks.

To conclude the functional validation, Subsubsection 8.2.2.3 is going to evaluate the impact of the dynamic parameters –which are supported within PREESM through SPiDER runtime manager– in the multiversioning system proposed in Chapter 5.

8.2.2.3 Dynamic performance evaluation

Once the combination of dataflow and polyhedral domains has been assessed through three different dataflow applications, the only study remaining is the analysis of how this combination reacts when some parameters change at runtime, considering that these parameters may affect the structure of the application and, thus, change the number of actors and their scheduling.

Since the objective is to highlight how Apollo adapts itself to this behavior, it is important to select an application that clearly shows the impact this polyhedral framework has on the overall time. As a result, it has been decided to choose Matmul application from the three used in the previous study, as it is the one in which the actor targeted by Apollo congregates more than a 95% of the overall execution time.

Figure 8-8 shows how the application graph evolves from being static –Figure 8-5– to being dynamic. As can be seen, all the changes are mainly related to parameters, and a new actor has appeared, *ModifyingSize*. All of them are marked with a white circle, which denotes that their behavior is dynamic –i.e., that they may change during the execution of the application–, and their objective is to modify and set at runtime the size of the matrices involved in the multiplication:

Figure 8-8: SPiDER application graph of Matmul, where dynamic parameters and actors are denoted with a white circle.

- First, *rowsA*, *colsA* and *colsB* set a default value for the dimensions of these matrices. It should be noted that there is no *rowsB* because, in order to multiply two matrices, the number of columns of the first matrix and the number of rows of the second must coincide (i.e., $colsA = rowsB$).
- *ModifyingSize* actor reconfigures, at the beginning of each iteration of the dataflow graph, the value of these parameters, and stores these values in the respective dynamic parameters: *rowsA_dyn*, *colsA_dyn* and *colsB_dyn*.
- These dynamic parameters are the ones considered as inputs for the rest of the actors, being this the main difference with respect to Figure 8-5, where the inputs of the actors were static parameters. As happened before, *rowsParallelized* still depends on the same parameters than in the static counterpart, but *rowsA* has been substituted by *rowsA_dyn*.

Thanks to these changes, now this dataflow application is dynamic in such a way that the problem size can be directly modified at runtime. Modifying the size of the matrices involved in the multiplication implies completely changing, for instance, the loop bounds of the loop nest performing the multiplication, so these changes should be definitely considered by a system such as Apollo to work properly: as observed in Table 8-17 and in Figure 8-6, for different problem sizes, different transformations are chosen by Apollo, and different behaviors are obtained in general; consequently, it is compulsory for the multiversioning system to detect these changes if they appear at runtime to configure itself accordingly, as described in Chapter 6.

The rest of the section is going to present the experiments conducted to evaluate the behavior of Apollo with the dynamic application presented in Figure 8-8 both in Namaka and Magerit. This application has been configured so the two phases of Apollo –training and operational– can be assessed for the sizes already defined for this application. To that end, the application switches between problem sizes each 10 consecutive iterations of the application graph, following the next process:

1. The default size is configured to the *Small* problem size, which generates matrices of 320×320 . At the beginning of the execution, the default size is maintained for 10 consecutive iterations of the application graph. With this configuration, Apollo is expected to invest all of them in the training phase.
2. *ModifyingSize* actor is configured to automatically change to *Large* size – 3200×3200 – at iteration 10, so the expected behavior is that Apollo detects this change and starts from scratch the training phase for the new values of the dynamic parameters. This behavior is again maintained during 10 iterations.
3. At iteration 20, *ModifyingSize* goes back to *Small*. The objective of this test is to show how Apollo retrieves the previous state and resumes the process where it was left, evaluating the remaining 5 transformations from the training set and automatically changing to the operational phase.

4. At iteration 30, the previous step is repeated again, with the objective of showing the equivalent behavior for the Large problem size.
5. At iteration 40, *ModifyingSize* changes back to Small. The objective of this test is to show how Apollo goes directly to the operational phase, since it has already performed the training phase for that specific value of the dynamic parameters.
6. At iteration 50, the size is changed back again to the Large size, to observe the same behavior than in the previous step, but for the other problem size.

Since the objective of this experiment is not characterizing for which applications is better to grant the parallelization to one tool or another, all the configurations used in the previous study –*PREESMPar*, *ApolloPar* and *CombPar*– are not going to be tested. It has already been proven that, for this application, the best configuration is *CombPar*, so this is the one that is going to be used for this study. As for the mode, as in the rest of the section, mode 4 –infinite timeout– will be maintained.

Table 8-18 gathers the execution times measured for each size and architecture. As can be observed, both the execution times and the transformations selected for each problem size and target architecture are similar to those obtained in the static experiment –Table 8-16–. These results are in line with what was expected, as the only difference between this application and the one analyzed in the previous study is the change of the parameters defining the size of the matrices, as they are now dynamic.

Table 8-18: Detail of the performance results obtained for Matmul application with SPiDER for both problem sizes and architectures (in seconds).

Size	Namaka			Magerit		
	Train	Oper	Tx	Train	Oper	Tx
Small	1.137	0.001	02	1.322	0.001	02
Large	5.922	1.322	09	6.198	1.137	09

To analyze the behavior of Apollo with respect to the dynamic parameters, Figure 8-9 provides a representation of how the execution time evolves with the iterations of the application graph for the specific case of Namaka. As can be observed, 4 different colors have been used to identify each phase of Apollo for each problem size: the training phase is represented in blue for the Small problem size and in yellow for the Large one; likewise, the operational phase is identified with orange for the Small size and with green for the Large one. When analyzing the graph, it should be noted that the y-axis is configured with a logarithmic scale so as to be able to represent together both problem sizes. Additionally, some numbers have been annotated to identify the moments in which the size is changed. With all this information, the information gathered in Figure 8-9 evolves as follows:

- When the application execution starts, the dynamic parameters are configured with the default problem size, so the training phase is started for the Small problem size ❶, which goes on for 10 iterations of the application graph.

Figure 8-9: Detail of the evolution of the execution time (in seconds) with respect to the iterations of the application graph.

- After iteration 10, a considerable increase of the execution time is observed ②. This increase is coherent, since *ModifyingSize* actor was configured to change between sizes each 10 iterations. Consequently, at iteration 10 this change is first produced, stopping the training phase of the *Small* problem size and beginning the training of the *Large* one, which is also maintained for 10 iterations.
- At iteration 20, the next change is produced, going back to *Small* ③. As can be observed, the training phase for this size is resumed. As only 5 transformations of the training set remained untested, at iteration 25 the training phase is completed, and the operational phase begins. This can be noticed (i) with the color change and (ii) with the drastic decrease in the execution time, which is now around 0.001 seconds, as shown in Table 8-18.
- At iteration 30, the size is reverted to *Large* ④, so its associated training phase is resumed. The behavior in these 10 iteration is equivalent to that observed in the previous ones: the training phase of the *Large* size finishes, and its operational phase begins, showing its corresponding decrease in the execution time, which falls to 1.1 seconds, approximately.
- Finally, at iterations 40 ⑤ and 50 ⑥, the subsequent changes in the size are produced, and, as can be seen, the respective operational phases are directly retrieved, without performing any further training, since *Apollo* successfully identifies those dynamic parameters as previously studied and automatically retrieves their last saved state.

With this last study, the functional validation of the methodology proposed in this manuscript is finally concluded. Since the beginning of this chapter, this functional validation has evaluated the changes and extensions addressed both in the polyhedral framework and in the dataflow tools, validating them and showing the advantages and disadvantages this combination may have. On the one hand, the validation from the polyhedral point of view was devoted to endorse the new extensions added within Apollo, proving their correctness through a set of benchmarks traditionally used in this domain. On the other hand, the validation from the dataflow point of view aimed at doing the same for the changes performed within both PREESM and SPiDER, but also at evaluating how to balance the parallelization possibilities both worlds offer, and showing how prioritizing one or the other may result in very different performances depending on the application.

Once this double validation has been addressed and analyzed, the only study remaining before closing this chapter is the analysis of how this new methodology can be used to automatically accelerate a real use-case application, as the one described in Section 7.2, and the advantages or disadvantages this methodology may have when compared to other implementations.

8.3 Use-case validation results

As established at the end of the previous section, once the methodology proposed in this document has been functionally validated from the point of view of dataflow and polyhedral domains, the only study remaining is to assess the performance this kind of methodology may provide to a real use-case application, as the one presented in Section 7.2. To do so, this section is going to be divided in two: first, Subsection 8.3.1 is going to present the implementation of the use-case in PREESM and SPiDER, together with the input images that are going to be considered for the evaluation; secondly, Subsection 8.3.2 is going to describe in detail the different experiments carried out for assessing the performance of the application; and, finally, Subsection 8.3.3 is going to provide the performance results obtained for this implementation with the combination of frameworks presented in this manuscript.

8.3.1 Use-case dataflow implementation

As aforementioned, the application used in this document as a real use-case has already been functionally described in Section 7.2. As explained there, this real use-case has been extracted from the customized classification framework developed in the context of HELICoiD project, which aimed at processing the hyperspectral images captured during a surgery to help the specialists in accurately delimiting the brain tumor being extirpated. To ease the readability of the document, the figure showing

the simplified block diagram of this classification framework, Figure 7-2, is repeated in Figure 8-10. It should be noted that the only difference between both images is that the part conforming the use-case is highlighted in Figure 8-10.

As can be observed, this hybrid classification framework is composed of 4 main blocks: preprocessing of the image, supervised classification, unsupervised classification and majority voting. As stated in [Fabelo'18b], the high computational load of this framework made compulsory offloading some parts to a hardware accelerator, namely the supervised classification stage, as it is the one demanding the most computational resources. For those reasons, the supervised classification approach has been chosen as the real use-case to validate the methodology presented in this work, as emphasized in Figure 8-10.

Figure 8-10: Block diagram of HELICoiD hybrid classification framework, where the part conforming the real use-case is highlighted.

During the development of HELICoiD project, the hardware accelerator chosen to offload the supervised classification stage was the Multi-Purpose Processor Array (MPPA-256-N) [Dinechin'13], a many-core architecture manufactured by Kalray S.A composed of 256 processing elements. The computational power offered by this accelerator made it suitable for this application, but it came at the cost of having to manually exploit the parallelization, which subsequently led to a wide range of studies and experiments until an implementation making an efficient usage of all the computational resources was devised [Lazcano'17c, Madroñal'17b, Florimbi'18, Martel'18, Lazcano'19a, Lazcano'19b].

To use this application as a real use-case, the implementations developed in the context of HELICoiD project have been taken as a basis to build the dataflow implementation using PREESM and SPiDER. Since the objective of this last study is to fully assess the methodology proposed in this document, only the implementation developed for SPiDER is going to be presented, as it is the one assessing all the new functionalities added to Apollo. This application presents two features that make it a good candidate for a dynamic dataflow implementation: first, the objective of this application is to process a stream of data while it is being generated in the operating room, and streaming applications are specifically well-suited for dataflow MoCs; secondly, the dynamism provided by SPiDER is especially interesting in this context, since the size of the images may evolve depending on the ROI, and it is

desirable that the application reconfigures itself if a change in the size of the images being processed is detected.

The rest of this section is organized as follows: first, Subsubsection 8.3.1.1 is going to analyze the dataflow implementation, getting into the details of each part of the supervised classification approach; afterwards, Subsubsection 8.3.1.2 will present the images that are going to be used as inputs for this implementation, which have been extracted from HELICoiD database [Fabelo'19b].

8.3.1.1 Implementation details

Taking advantage of the characteristics of PiSDF MoC, the dataflow implementation of HELICoiD supervised classification stage has been built as a hierarchical application, as it helps in building the implementation following a top-down approach, which in turn allows encapsulating the different peculiarities and characteristics in different levels. Figure 8-11 gathers the top-level application graph, composed by 4 actors and 10 parameters, almost all of them dynamic.

Figure 8-11: Top-level SPiDER application graph of HELICoiD supervised classification stage.

With respect to the actors, two of them are related to Input/Output (I/O) operations, since they are in charge of reading the input image –*Read_HS_image*– and saving the results –*Write_results*–; as for the other two, *HSI_cancer_detection* is a hierarchical actor containing the rest of the implementation –it should be noted that a white background in an actor indicates that it is a hierarchical actor–, while *ConfigSelector* is the dynamic actor in charge of detecting changes in the image dimensions and reconfiguring the dynamic parameters accordingly.

As for the parameters, the reason why almost all of them are dynamic –as a reminder, a dynamic parameter is identified with a white circle– is that, as already mentioned, the dynamism in the application comes from the possibility of modifying the image size at runtime; hence, all the parameters depending on the image dimensions must be dynamic, namely *bands*, *rows* and *columns*, which denote the hyperspectral image dimensions, and *imageBeingProcessed*, which provides an identifier of the image being processed. The rest of the dynamic parameters are so because they depend on the image dimensions to some extent; as shown in Figure 8-11, all of them are related to the parallelization of the different stages of the supervised classification –*parallelismCovarianceMatrixA*, *parallelismImageCentering*, *parallelismProjection*, *ParallelismSVM* and *ParallelismKNN*–. Finally, the only static parameter is *numClasses*, which represents the number of classes the supervised classifier must identify. The functionality of these parameters will be further analyzed once the parts of the implementation influenced by them are described.

Figure 8-12 presents the application graph associated to the *HSI_cancer_detection* actor. As can be observed, this graph contains 5 new actors, while the parameters are the same than those appearing in Figure 8-11. It should be noted that the application graph shown in Figure 8-12 has been simplified to ease the readability of the figure: as can be observed, all the dynamic parameters of the application appear in the top-level application graph –Figure 8-11–, so they need to be inherited in the subsequent graphs to be able to use them. This implies the appearance of a specific actor that matches the incoming values with a local copy in a parameter of the corresponding graph, hence multiplying the number of connections and increasing the complexity of the figure.

Figure 8-12: SPiDER application graph of the *HSI_cancer_detection* actor shown in Figure 8-11.

Figure 8-13 provides an example of an instance of this specific actor, showing the behavior of only one parameter –*bands*– for illustration purposes. As can be seen, the inputs are the parameters shown in Figure 8-11, and the outputs are connected to the local copies of the parameters. This mechanism is needed in every level of the hierarchy, so hereafter it will be automatically omitted for the rest of the levels.

Figure 8-13: Detail of the actor specifically devised for passing the top-level parameters to lower levels of the hierarchy.

Regarding the new actors, it can be seen that three are hierarchical again, namely *SupervisedClassification*, *DimensionalityReduction* and *SpatialSpectralFiltering*, while the other two are actors specifically devoted to adapt somehow the incoming data: *BroadcastHsImage* is a special actor that broadcasts the input image to the actors using it, and *TransposeMatrix* is in charge of transposing the input image so as to comply with the data layout required by the actor connected to it. Regarding the hierarchical actors, each of them contains the application graph of each of the algorithms composing the supervised classification approach, i.e., PCA, SVM and KNN, respectively. The functionality of these algorithms was already described in Section 7.2, so here the focus will be on describing the implementation of each one.

As a last remark before analyzing the hierarchical actors, it is worth pointing out that, if one compares the application graphs gathered in Figure 8-11 and Figure 8-12 with the block diagram presented in Figure 8-10, one of the advantages of the dataflow paradigm is highlighted. As can be seen, the application graph shown in Figure 8-11 can be clearly identified with the *Supervised classification* block; likewise, the graph of Figure 8-12 represents the structure of how PCA, SVM and KNN are interconnected. This comparison shows that the dataflow paradigm is very useful to develop applications in a modular and hierarchical way. Now, the hierarchical actors gathering the implementations of SVM, PCA and KNN are going to be analyzed.

SVM

Figure 8-14 shows the application graph associated to the *SupervisedClassification* actor, which contains the implementation of SVM. As described in Section 7.2, SVM algorithm is decomposed in two different stages: training and classification. The first one generates a model that is afterwards used by the classification stage; however, in the context of HELICoiD project, the training is performed offline, so the model is previously generated and must be read. For this reason, this graph contains again a hierarchical actor, so it can be considered the top-level graph of SVM algorithm. At this level, the main task is reading the aforementioned model, which is carried out by *ReadSvmModel* actor. This actor reads the model from memory and generates the five inputs required, which are then fed to *SVM*, a hierarchical actor containing the implementation of the classification stage.

Before getting into the details of *SVM* actor, it can be seen that there are 5 actors colored in orange between *ReadSvmModel* and *SVM* actors, one per output generated

Figure 8-14: SPiDER application graph of the *SupervisedClassification* actor included in Figure 8-12.

by the former. These are special actors representing round buffers, a mechanism required by SPiDER to make possible the parallelization of this graph. For that reason, one of the input parameters of all of them is *parallelismSVM*: through this parameter, users can specify how many instances of *SVM* actor they wish to execute in parallel, and these special actors make sure that their corresponding input data arrives to each instance of the actor. This is the reason why *TransposeMatrix* actor was needed between the input image and the *SupervisedClassification* actor in Figure 8-12: as already stated in Section 7.2, SVM is a pixel-wise classifier, which means that all the processing involved is performed at pixel-level, and, therefore, that it can follow a pixel-wise parallelization strategy. Since the input image is read band by band, before arriving to SVM, it needs to be reordered, so afterwards it can be directly parallelized by dividing the input image in as many slices as number of instances users want to define. As for the round buffers, their role in the implementation is to replicate the model as many times as instances are defined to make possible the parallelization, which is why they are placed between the reading of the model and the classification stage.

To conclude the description of SVM implementation, Figure 8-15 provides the graph associated to the *SVM* actor included in Figure 8-12. As aforementioned, this hierarchical actor contains the implementation of the classification stage. In Section 7.2, this stage was said to be composed of three successive phases, all of them following a pixel-wise procedure: (i) computation of both the distance of each pixel to each binary hyperplane and the associated binary probabilities; (ii) combination of the binary probabilities following the procedure described in [Wu'04] to minimize the complexity of the implementation; and (iii) selection of the class with the highest probability to generate the classification map. However, in the context of HELICoiD

project, SVM is coupled with a spatial filtering; as a result, the last phase is not needed, since this process will be carried out after this filtering step. Consequently, as shown in Figure 8-15, the dataflow implementation of the SVM classification stage has been divided into 2 actors, each one taking care of one of the phases mentioned above: *PairwiseProbsComputation* for the first, and *MinimizationSolver* for the second. As mentioned before, *SVM* actor can be directly parallelized by slicing the inputs so each slice –containing a block of pixels– is sent to a different instance of this actor. Hence, creating different instances of *SVM* actor implies, in turn, creating different instances of the application graph shown in Figure 8-15.

Figure 8-15: SPiDER application graph of the *SVM* actor included in Figure 8-14.

Algorithm 8-1 gathers the pseudocode of the classification stage. As aforementioned, this pseudocode has been extracted from the implementations developed and refined in the context of HELICoiD, so more information can be found in [Lazcano'19b] and [Madroñal'17b], where the application of the methodology proposed in [Wu'04] to solve the minimization problem is detailed.

This pseudocode is composed of two sequential for-loops, each one representing the operation of each actor appearing in Figure 8-15. As can be seen, each iteration of each loop processes a block of pixels; hence, it will involve more or less computation depending on the size of that block.

Regarding *PairwiseProbsComputation* actor, as can be observed, the first step is to compute the distances of each pixel to the different model hyperplanes –line 4–, which are generated offline during the training phase and identify the regions maximizing the distance between each pair of classes. Once the distances are computed, the binary probabilities associated to each pixel are calculated –line 6–, representing the probability of each pixel to belong to each of the two classes separated by the corresponding hyperplane. With this step, this actor finishes its processing,

Algorithm 8-1 SVM classification stage

Input: *transposedImage*, *svmModel***Output:** *pixelsProbs*

```
1: //PairwiseProbsComputation actor
2: for all pixelBlock in transposedImage do
3:   //Compute distances to hyperplanes
4:   pixelDistancespixelBlock = pixelBlock * modelhyperplanes
5:   //Compute binary probabilities
6:   pairwiseProbspixelBlock = pairwiseProbs(pixelDistancespixelBlock, modelprobs)
7: end for
8: //MinimizationSolver actor
9: for all pixelBlock in HITransposed do
10:  //Compute pixel probabilities
11:  pixelsProbspixelBlock = minimization(pairwiseProbspixelBlock)
12: end for
```

generating the output shown in Figure 8-15 –*pairwiseProb*–, and forwarding it to *MinimizationSolver* actor. This actor applies the mentioned procedure to minimize the computational complexity of combining the binary probabilities –line 11.

PCA

Figure 8-16 shows the application graph associated to the *DimensionalityReduction* actor, which contains the implementation of PCA. As described in Section 7.2, PCA algorithm is divided into four successive stages: (i) image centering, which consists of computing and removing the average of each band of the input image; (ii) covariance matrix computation, which basically consists of multiplying the input image by itself; (iii) eigenvector extraction, which is performed by means of Jacobi algorithm; and (iv) projection, which consists of projecting the centered image onto the eigenvectors subspace. These stages can be clearly identified in Figure 8-16, since there is one actor per stage: *ImageCentering*, *Covariance*, *Jacobi* and *Projection*.

As for the rest of the actors appearing in this figure, *TransposeImageCentered* deals again with reshaping a matrix to match the data layout required by *Projection*, and the rest of them are either broadcasts or round buffers, whose functionality is still making possible the parallelization of the image. This parallelization is defined by the parameters, but, contrary to SVM, in this case the parallelization is not applied to the whole graph, but independently configured for each stage of the algorithm. In the following paragraphs, an overview of each actor is provided, but more information about the implementations can be found in [Lazcano'19b].

In the case of *ImageCentering*, as can be observed, it is a hierarchical actor, and its associated application graph is shown in Figure 8-17. When describing this step, one could easily see that it is in turn composed of two sequential steps: (i) computing the average of each band of the input image and (ii) removing this average from

Figure 8-16: SPIDER application graph of the *DimensionalityReduction* actor included in Figure 8-12.

each band of the image, which can be identified with *MeanValueComputation* and *MeanValueRemoval* actors, respectively. From the tasks carried out by *ImageCentering*, it can be easily inferred that this stage is band-wise parallelizable, since the operations can be independently carried out for each band of the image. Consequently, by tuning *parallelismImageCentering* parameter, one can decide in how many blocks of successive bands the input image is divided, so that they can be processed in parallel. Algorithm 8-2 presents the pseudocode associated to the two actors composing *ImageCentering*. As happened with SVM, the first nested for-loop corresponds with the operation of *MeanValueComputation* actor, while the second one gathers the processing of *MeanValueRemoval*.

Figure 8-17: SPiDER application graph of the *ImageCentering* actor included in Figure 8-16.

Algorithm 8-2 *Image centering stage of PCA algorithm*

Input: image, pixels, bands

Output: imageCentered

```

1: //MeanValueComputation actor
2: for all bands in imageblock do
3:   //Compute the average of each band
4:   for all pixels in this band do
5:     average += accumulateValues(imageblock)
6:   end for
7:   average = average / pixels
8: end for
9: //MeanValueRemoval actor
10: for all bands in imageblock do
11:   //Center each band
12:   for all pixels in this band do
13:     imageCenteredblock = removeAverage(imageblock, average)
14:   end for
15: end for
  
```

The pseudocode associated to *Covariance* actor is presented in Algorithm 8-3. As already mentioned, this actor is in charge of multiplying its input image, *imageCentered*, by its transpose to generate the corresponding covariance matrix. Consequently, this operation involves a traditional matrix multiplication, which has

been modeled to mimic the parallelization strategy followed by Matmul benchmark –Subsection 8.2.2–. Therefore, users can control the level of parallelism through *parallelismCovarianceMatrixA*, which defines the number of blocks of consecutive bands –*bands_{block}*– in which the input matrix is going to be divided. Then, each of these blocks can be processed separately, since multiplying the different blocks by the input matrix yields different blocks of the resulting covariance matrix.

Algorithm 8-3 *Covariance* stage of PCA algorithm

Input: imageCentered, bands

Output: cov

```

1: //For each row of the resultant matrix
2: for band1 = 0 to band1 = bandsblock do
3:   //For each column of the resultant matrix
4:   for band2 = 0 to band2 = bands do
5:     //2-vector multiplication
6:     covband1band2 = imageCenteredband1 * imageCenteredband2
7:   end for
8: end for

```

Likewise, Algorithm 8-4 gathers the pseudocode of *Jacobi* actor. The input of this algorithm is the covariance matrix generated in the previous step, *cov*, and the output is a matrix with the first *PCs* eigenvectors, being *PCs* the number of principal components that must be calculated. As already described, Jacobi is an iterative process in which a matrix is diagonalized by zeroing one off-diagonal element at a time until all of them are zeroed or a stop condition is fulfilled; hence, at each iteration of the algorithm, one off-diagonal element of the matrix is zeroed –line 4–, and its associated rows and columns are updated –lines 6-7–. This process goes on until all the elements have been zeroed or a stop condition is met; in this case, the stop condition is met if, after a certain number of iterations, the convergence is still not reached. Once the iterative process finishes, the eigenvectors are reordered as a function of the value of their associated eigenvalues –lines 11-13–, which can be found in the diagonal of the matrix –*diagonalMatrix*–. Finally, the algorithm returns the first *PCs* eigenvectors; in the context of HELICoiD, this number has been set to 1, since the objective of PCA is to provide a one-band representation of the spectral information of the image.

As shown in Figure 8-16, there is no parallelization parameter for *Jacobi*: since this algorithm is an iterative process that is not completed until, at least, one of two conditions are met, the number of iterations cannot be predicted in advance. This behavior cannot be modeled in PREESM, and thus the structural parallelism of this actor cannot be directly exploited. Nonetheless, a deep analysis of this algorithm, together with a proposal to manually parallelize it, can be found in [Lazcano'17c].

Finally, the pseudocode associated to *Projection* actor is provided in Algorithm 8-5. As mentioned before, the objective of PCA algorithm within HELICoiD project is to generate a one-band representation of the image, so only the projection onto the

Algorithm 8-4 *Jacobi* stage of PCA algorithm

Input: cov, PCs**Output:** eigenvector

```
1: diagonalMatrix = cov
2: //Do the diagonalization
3: while !isDiagonal(diagonalMatrix) and !stopCondition do
4:   valuesBeingZeroed = getZeroedValued(diagonalMatrix)
5:   for all value in valuesBeingZeroed do
6:     [diagonalMatrix, eigenvectorMatrix] = updateRows(diagonalMatrix, value)
7:     [diagonalMatrix, eigenvectorMatrix] = updateCols(diagonalMatrix, value)
8:   end for
9: end while
10: //Reorder the eigenvectors
11: eigenvalues = getDiagonal(diagonalMatrix)
12: eigenvaluesOrdered = order(eigenvalues)
13: eigenvectors = reorder(eigenvectorMatrix, eigenvaluesOrdered)
14: //Return the first PCs eigenvectors
15: eigenvector = selectFirstEigenvector(eigenvectorMatrix, PCs)
```

first eigenvector is required. This feature considerably reduces the complexity of this stage, since the problem goes from multiplying two matrices (i.e., projecting the centered image onto all the eigenvectors) to only multiplying a matrix by a vector, the one containing the first eigenvector. Anyhow, if users still consider necessary the parallelization of this stage, it can be easily inferred that it can follow a pixel-wise parallelization strategy: the input image, *imageCenteredTransposed*, is organized pixel by pixel, and each pixel can be independently multiplied by the eigenvector. As a result, *parallelismProjection* configures this parallelization in such a way that, as with SVM, it denotes the number of slices in which *imageCenteredTransposed* is going to be divided, each *pixelBlock* thus containing a certain number of successive pixels. By setting this parameter to the number of PE available, each slice can then be processed in a different PE simultaneously.

Algorithm 8-5 *Projection* stage of PCA algorithm

Input: imageCenteredTransposed, eigenvector**Output:** pcaOutput

```
1: for all pixelBlock in imageCenteredTransposed do
2:   //Do the projection
3:   pcaOutputpixelBlock = eigenvector * pixelBlock
4: end for
```

KNN

Finally, Figure 8-18 shows the application graph associated to the *SpatialSpectralFiltering* actor, which contains the implementation of KNN. From the description of KNN provided in Section 7.2, it can be inferred that it is divided into three consecutive stages: (i) computing the distances of each pixel to every other pixel in the image; (ii) generating as many probability maps as number of classes must be discerned by the supervised classifier –parameter *numClasses* from Figure 8-11–; and (iii) generating a final classification map by assigning to each pixel the label of the class with the highest probability. These three steps can be clearly identified with three actors of the application graph shown in Figure 8-18, namely *KnnSearch*, *Filtering* and *DecisionMaking*, respectively. Before these actors, however, there is another one, *featMatBuilding*, which is in charge of preparing the data so the correct information arrives to *KnnSearch*. To compute the distances, three different pieces of information are required for each pixel in the image: the associated spectral value –coming from PCA output– and two spatial coordinates to identify the row and the column where the associated pixel is located. Consequently, before starting the computation, a matrix with these three values per pixel –*featMat*– is built.

Figure 8-18: SPiDER application graph of the *SpatialSpectralFiltering* actor included in Figure 8-12.

As mentioned in Section 7.2, KNN algorithm depends on two configuration parameters, K and λ . The former denotes the number of neighbors KNN algorithm needs to find for every pixel in the image, while the latter balances the weight of the spatial coordinates and the spectral value of each pixel. Within HELICoiD project, these values were set to 40 and 1, respectively, as it has been proven that choosing larger values may result in over-smoothing the classification result [Fabelo'18b]. Consequently, for every pixel in the image: (i) 40 different neighbors need to be found, and (ii) its associated spatial and spectral information is equally balanced ($\lambda = 1$).

Additionally, [Florimbi'18] proved also the advantages of defining a search window for the neighbor selection, which is defined as a region with such a size that it encloses all the neighbors of a certain pixel. By doing so, the search of the neighbors is limited to the size of this window, not to the entire image, thus resulting in an important reduction of the computational complexity. After a thorough analysis of the size of

this window, the value selected was 14 –i.e., 14 consecutive rows of the image–, since it is the one guaranteeing the same classification results than those obtained when considering all the image for finding the neighbors.

Consequently, this window approach is also applied in the dataflow implementation. As can be observed, there is a parameter, *windowSizeKnn*, which defines the number of rows conforming the window –as already established, 14–; this parameter, together with *parallelismKNN* –and, more specifically, *pixelsParallelism*–, is used to properly define the parallelism of this algorithm. As already said, it can be parallelized at the pixel-level; however, exploiting its structural parallelism is not as straightforward as directly splitting the image in equal slices and processing them separately; due to the window mechanism, each pixel requires its complete search window to be correctly processed, and the search window is different for each pixel. To illustrate how this mechanism works, Figure 8-19 provides a toy example where the search window is represented for one pixel –Figure 8-19a–, for N consecutive pixels –Figure 8-19b– and for the next N consecutive pixels –Figure 8-19c.

Figure 8-19: Example of the search window mechanism, where blue represents the pixels to be processed, green identifies the starting point of the window, and yellow the ending point.

As can be observed, if a window size of 14 rows is defined, this window starts 7 rows immediately before the pixel under study –in the figure, represented in green–, and finishes 7 rows after –represented in yellow–. When this mechanism is extended so as to process N consecutive pixels, the window required is the one shown in Figure 8-19b. Hence, to process a slice containing N consecutive pixels, the information the slice must contain follows Equation 8-5.

$$slice = windowSizeKnn \times cols + N \quad (8-5)$$

Additionally, when processing the next N pixels, as shown in Figure 8-19c, it can be seen that the new window is deeply overlapped with the previous one –Figure 8-19b; as a result, to split the image and parallelize the searching step, the input data –*featMat*– must be previously adapted: in the frontiers of each slice, the information belonging to successive windows must be duplicated, so this new *featMat* can now be split homogeneously. This process is automatically performed by *featMatBuilding*.

With these changes, *featMat* will be split in as many blocks as *parallelismKNN* indicates; likewise, the same number of instances of *KnnSearch* will be created, so each one finds all the neighbors related to the pixels included in one block. The pseudocode associated to this actor is provided in Algorithm 8-6; as can be observed, there is a searching for each pixel included in a block of pixels –line 4–, in charge of finding the desired neighbors –in this case, 40.

Algorithm 8-6 *Searching stage of KNN algorithm*

Input: *featMat*

Output: *knnMatrix*

```

1: for all pixelBlock in featMat do
2:   //Search neighbors of each pixel
3:   for all pixel in pixelBlock do
4:     knnMatrixpixel = searching(pixel, window, neighbors)
5:   end for
6: end for

```

Finally, the pseudocodes associated to *Filtering* and *DecisionMaking* actors are gathered in Algorithm 8-7 and Algorithm 8-8, respectively. As can be observed, the same parallelization strategy has been followed: each instance of *KnnSearch* actor generates a block of *knnMatrix*, which contains all the neighbors associated to each pixel in that block; subsequently, this *knnMatrixBlock* can be processed by one instance of *Filtering* actor, thus creating again as many instances as *knnMatrixBlocks*, and the same happens again with *DecisionMaking* actor. *Filtering* is in charge of averaging the probability of all the neighbors to belong to each one of the classes defined by *numClasses* parameter –lines 4-7–, while *DecisionMaking* deals with selecting the class with the highest probability, and thus generating the final output of the supervised classification approach –line 3.

Algorithm 8-7 *Filtering stage of KNN algorithm*

Input: *knnMatrix*, *pixelProbs*

Output: *knnProbs*

```

1: for all pixel in knnMatrixBlock do
2:   for all numClasses do
3:     //Average the probability
4:     for all neighbors do
5:       knnProbspixelclass += pixelProbsneighborclass
6:     end for
7:     knnProbspixelclass = probsMappixelclass / neighbors
8:   end for
9: end for

```

Algorithm 8-8 *Decision making stage of KNN algorithm***Input:** knnProbs**Output:** finalClassMap

```

1: //Select the class with the highest probability
2: for all pixel in knnMatrixBlock do
3:   finalClassMappixel = maxProbability(knnProbspixel)
4: end for

```

8.3.1.2 Input images

Once the dataflow implementation of HELICoiD supervised classification stage has been fully described, the only feature remaining to completely characterize the real use-case is to introduce the images that are going to be used to evaluate it. As already mentioned, one of the outcomes of HELICoiD project was the creation of an open-source database containing 36 hyperspectral images captured in-vivo during surgical procedures and showing brain surfaces, where a tumor may –or not– be exposed [Fabelo’19b]. Table 8-19 presents the three images that have been selected among those 36 to evaluate the implementation. They have been selected because they present different spatial resolutions –as can be inferred from *Rows* and *Columns* values in the table–, and they have already been used to assess some of the implementations developed within the context of HELICoiD [Lazcano’19b]. It is worth noting that these images are the result of a preprocessing stage, in which those bands with too low Signal-to-Noise Ratio (SNR) have been completely removed, and many others have been averaged and removed, following the procedure described in [Fabelo’16a].

As can be observed, each image shows a brain surface, and two black rubber markers stand out. It is worth pointing out that these markers were placed by surgeons to mark two opposite features: in all the images, one marker identifies the parts where they believed the tumor was placed, while the other defines a region where no tumor is appreciated. The objective of placing these markers was to take samples from within each marker to use them as gold references; nonetheless, the process of extracting the gold references evolved as described in [Fabelo’19b].

Table 8-19: HSI brain cancer images used during the validation.

Brain images
Name	Brain-1	Brain-2	Brain-3
Rows	442	375	377
Columns	496	493	329
Bands	128	128	128

8.3.2 Specific setup

Once both the dataflow implementation of the real use-case and the input images have been described, the rest of the chapter is devoted to analyzing the performance results achieved for this application with the methodology proposed in this document.

However, before starting with the analysis of this study, the experiments that have been carried out must be outlined. Since the objective is to test the advantages and disadvantages of a methodology like the one proposed here, but with a real use-case application, the structure followed for this experiment is similar to the previous ones:

- First, the sequential version –based on Clang– will be assessed. This time, since the application simulates a real-life scenario, the functional results obtained will be shown to illustrate the expected output of the supervised classification stage.
- After that, the rest of the configurations will be assessed: *PREESMPar*, *ApolloPar* and *CombPar*. The first one explores the structural parallelism through PREESM; the second assesses the potential of Apollo to parallelize this application; finally, the last one analyzes the combination of both frameworks. The parallelization potential will be assessed, as in the rest of the studies, using Namaka and Magerit architectures, with 16 and 20 PEs, respectively. Likewise, following the structure of the rest of the chapter, the infinite timeout will be used again for Apollo, as the objective is to maximize the impact in the performance the tool may have.

With this study, the assessment of the new methodology will be fully completed, having enough evidence to weigh its advantages and disadvantages.

8.3.3 Performance results

As established in Subsection 8.3.2, the first study that is going to be carried out is the analysis of the sequential times. The objective of this study is twofold: first, to functionally assess the results obtained with the supervised classification approach; secondly, to characterize the application in terms of execution times, so as to locate the bottlenecks of the dataflow application to identify which actors should be parallelized.

On the one hand, as for the functional assessment, Table 8-20 gathers the original images and their associated outputs generated by each of the algorithms composing the supervised classification stage. As can be observed, this stage generates maps where 4 different classes are identified: tumor tissue (red), healthy tissue (green), hypervascularized tissue (blue) and other materials (black). This configuration has been chosen because it is the one used in HELICoiD project. Although the markers were the initial mechanism to identify tumor and healthy samples, these images serve to prove the benefits of applying HSI to human brain cancer detection, as they have been validated by the neurosurgeons in the context of HELICoiD project:

- Regarding Brain-2, it can be seen that, as expected, the inside of one marker corresponds with tumor tissue, while the other is labeled as healthy tissue.

- However, for Brain-3 and Brain-1, this hypothesis is not held: as for the former, one of the markers does not contain any tumor tissue, but the other contains a mix of tissues, since both healthy and tumor tissues can be found within the marker; in the case of the latter, it is even worse, since none of the markers contains tumor samples. Anyhow, both cases identify a problem: depending on the surgeon's naked eye to identify tumors is a prone-error procedure, since it is difficult, even for specialists, to discern where the tumor ends and the healthy tissue begins, which was the original idea upon which HELICoiD was built.

Table 8-20: PCA, SVM and KNN results obtained for the dataflow implementation using the images presented in Table 8-19.

Name	Image	PCA	SVM	KNN
Brain-1
Brain-2
Brain-3

On the other hand, regarding the performance characterization, Table 8-21 presents a detailed analysis of the execution time associated to each actor of the dataflow implementation, together with the workload each actor represents over the total time required to execute the whole application.

With these numbers, the bottlenecks of each algorithm can be easily identified; afterwards, this information will be used to configure the parallelization, targeting the most time-consuming actors. It should be noted that actors in charge of I/O operations, configuration actors and special actors have been left out of this table, since they cannot be parallelized and, thus, that information is somehow redundant. Yet, these times are considered in the total execution time, which is why this number and the addition of the execution times of the actors may differ. The same also applies to *TransposeMatrix* actor in Figure 8-12; since this actor is not included in PCA, SVM or KNN, and there is already another actor transposing an image of the same dimensions

8.3. USE-CASE VALIDATION RESULTS

Table 8-21: Performance results obtained for the sequential execution of the use-case (in seconds) and workload associated to each actor (in %).

Stage		Actor	Namaka		Magerit	
			Time	Workload	Time	Workload
Brain-1	PCA	<i>MeanValueComputation</i>	0.023	0.3 %	0.033	0.5 %
		<i>MeanValueRemoval</i>	0.026	0.3 %	0.043	0.4 %
		<i>TransposeImageCentered</i>	0.259	3.3 %	0.428	4.0 %
		<i>Covariance</i>	3.119	39.5 %	4.930	46.1 %
		<i>Jacobi</i>	0.007	0.1 %	0.007	0.1 %
		<i>Projection</i>	0.023	0.3 %	0.037	0.3 %
	SVM	<i>PairwiseProbsComputation</i>	0.149	1.9 %	0.218	2.0 %
		<i>MinimizationSolver</i>	0.037	0.5 %	0.050	0.5 %
	KNN	<i>featMatBuilding</i>	≈ 0	≈ 0 %	0.002	≈ 0 %
		<i>KnnSearch</i>	3.964	50.2 %	4.541	42.5 %
		<i>Filtering</i>	0.027	0.3 %	0.037	0.3 %
		<i>DecisionMaking</i>	0.001	≈ 0 %	0.001	≈ 0 %
	TOTAL		8.008	100 %	10.695	100%
Brain-2	PCA	<i>MeanValueComputation</i>	0.019	0.3 %	0.028	0.4 %
		<i>MeanValueRemoval</i>	0.021	0.3 %	0.036	0.4 %
		<i>TransposeImageCentered</i>	0.218	3.3 %	0.358	4.0 %
		<i>Covariance</i>	2.599	39.3 %	4.137	46.1 %
		<i>Jacobi</i>	0.006	0.1 %	0.007	0.1 %
		<i>Projection</i>	0.019	0.3 %	0.031	0.4 %
	SVM	<i>PairwiseProbsComputation</i>	0.125	1.9 %	0.184	2.1 %
		<i>MinimizationSolver</i>	0.030	0.5 %	0.040	0.4 %
	KNN	<i>featMatBuilding</i>	≈ 0	≈ 0 %	0.001	≈ 0 %
		<i>KnnSearch</i>	3.329	50.4 %	3.811	42.5 %
		<i>Filtering</i>	0.022	0.3 %	0.031	0.3 %
		<i>DecisionMaking</i>	0.001	≈ 0 %	0.001	≈ 0 %
	TOTAL		6.757	100%	8.976	100%
Brain-3	PCA	<i>MeanValueComputation</i>	0.013	0.3 %	0.019	0.5 %
		<i>MeanValueRemoval</i>	0.014	0.3 %	0.024	0.4 %
		<i>TransposeImageCentered</i>	0.147	3.7 %	0.240	4.4 %
		<i>Covariance</i>	1.744	43.8 %	2.754	50.4 %
		<i>Jacobi</i>	0.005	0.1 %	0.006	0.1 %
		<i>Projection</i>	0.013	0.3 %	0.022	0.4 %
	SVM	<i>PairwiseProbsComputation</i>	0.084	2.1 %	0.124	2.3 %
		<i>MinimizationSolver</i>	0.021	0.5 %	0.028	0.5 %
	KNN	<i>featMatBuilding</i>	≈ 0	≈ 0 %	0.001	≈ 0 %
		<i>KnnSearch</i>	1.781	44.7 %	2.014	36.9 %
		<i>Filtering</i>	0.015	0.4 %	0.021	0.4 %
		<i>DecisionMaking</i>	≈ 0	≈ 0 %	≈ 0	≈ 0 %
	TOTAL		4.086	100%	5.460	100%

in PCA –*TransposeImageCentered*–, it has not been included as an independent item in Table 8-21, but it is considered in the total execution time, and the optimizations applied to it will be the same than those applied to *TransposeImageCentered*.

As can be observed, there are two actors accumulating most of the execution time regardless of the input image and the target platform: *KnnSearch* and *Covariance*. Between both of them, they gather near 90% of the total execution time; hence, these actors are the ones that can benefit more from the parallelization possibilities, so they must be targeted by either tool or by the combination of them.

As a last remark, a feature drawing attention is that, in the sequential version, Magerit takes considerably longer than Namaka; this is due to the difference in the operating frequencies, since Magerit runs at 2.1GHz and Namaka at 3.4GHz.

Once the sequential implementation has been characterized in terms of performance and the bottlenecks have been detected, next step is to exploit the parallelism through PREESM. To that end, now that the input images –and, more importantly, their dimensions– are defined, the different levels of parallelism need to be configured using PREESM parameters for Namaka and Magerit.

Table 8-22 gathers the parameters involved to some extent in the parallelization of the application for both Namaka and Magerit, together with the values they will take for the experiments in which PREESM parallelizes –i.e., *PREESMPar* and *CombPar*– and other parameters they depend on. As can be observed, there is also a number between parentheses, which represents the size of the resulting slice to be processed.

- The first two parameters indicate a size of 8; this means that the input of the stages they refer to –i.e., *ImageCentering* and *Covariance*, whose input is a matrix of the same dimensions for both of them– is divided into 16 slices of 8 bands each, thus following a band-wise parallelization strategy. As the number of bands is the same for the three input images, this parallelization is equal for all of them.
- On the contrary, the rest of the parallelization parameters depend either on the number of pixels or the number of rows, which differ from one image to another; hence, the size of the slices varies among images.

Table 8-22: Configuration of the parameters related to the parallelization of the application.

Parameter name	Dependence	Brain-1	Brain-2	Brain-3
<i>ParallelismImageCentering</i>	bands	16 (8)	16 (8)	16 (8)
<i>ParallelismCovarianceMatrixA</i>	bands	16 (8)	16 (8)	16 (8)
<i>ParallelismProjection</i>	pixels (<i>rows</i> × <i>columns</i>)	26 (8432)	25 (7395)	29 (4277)
<i>ParallelismSVM</i>	pixels (<i>rows</i> × <i>columns</i>)	26 (8432)	25 (7395)	29 (4277)
<i>ParallelismKNN</i>	pixels (<i>rows</i> × <i>columns</i>)	26 (8432)	25 (7395)	29 (4277)

Regarding the number of slices configured for each stage, since they depend on the number of bands or pixels of the input image, they have been selected among the possible divisors of these numbers. The rationale behind this choice is twofold:

- With respect to the parameters depending on the number of bands, the parallelism level that has been chosen is 16. As already established, the number of bands is always $128 - 2^7$; consequently, all the divisors are power of 2. In this case, the divisor chosen has been 16 because it is a good compromise between choosing a parallelization level close to the number of available PEs and avoiding the over-atomization of the actor: as can be observed in Table 8-22, with a level of 16, the resulting slices are composed of 8 bands; for example, in the *Covariance* actor, this means that the kernel computing the covariance will have an outermost loop bound of 8. Since Apollo requires some iterations to instrument the code and build the prediction model, if this value is decreased even more, the kernel will not have enough workload for Apollo to be successful. Consequently, the value of 16 has been set for both architectures.
- Regarding the parameters depending on the number of pixels, the parallelism level has been chosen to be close to the number of available PEs for each architecture –i.e., 16 and 20, respectively–, so that SPiDER can perform efficient workload distributions, so no PE is ever idle.

Once the parameters handling the parallelization in PREESM have been configured, the last step before evaluating the performance results achieved for the three configurations –PREESMPar, ApolloPar and CombPar– is to analyze if the actors can be targeted by Apollo or not, and, if not, justify why, so as to try to either find a solution –if it is straightforward– or leave it as a possible future line.

To that end, Table 8-23 gathers the actors composing the dataflow application and their average workload, together with a ✓ if the actor is targeted by Apollo, or a ✗ if not; in this last case, column *Reason* states why that actor is not compliant with Apollo.

Table 8-23: Actor average workload (in %) and possibility to be targeted by Apollo.

Stage	Actor	Workload	Apollo?	Reason
PCA	<i>MeanValueComputation</i>	$\approx 0.4\%$	✓	–
	<i>MeanValueRemoval</i>	$\approx 0.4\%$	✓	–
	<i>TransposeImageCentered</i>	$\approx 4.0\%$	✓	–
	<i>Covariance</i>	$\approx 45.0\%$	✓	–
	<i>Jacobi</i>	$\approx 0.1\%$	✗	Negligible workload
	<i>Projection</i>	$\approx 0.4\%$	✓	–
SVM	<i>PairwiseProbsComputation</i>	$\approx 2.0\%$	✗	No prediction model
	<i>MinimizationSolver</i>	$\approx 0.5\%$	✗	No prediction model
KNN	<i>featMatBuilding</i>	$\approx 0\%$	✗	Negligible workload
	<i>KnnSearch</i>	$\approx 45.0\%$	✗	No prediction model
	<i>Filtering</i>	$\approx 0.3\%$	✗	No prediction model
	<i>DecisionMaking</i>	$\approx 0\%$	✗	Negligible workload

In the case of PCA, all actors but *Jacobi* are targeted by Apollo, and this one is not targeted because, as can be noticed, its workload is near 0 %, so it does not seem worth it trying to parallelize it. The same applies to *featMatBuilding* and *DecisionMaking*:

their workload is so small that applying the multiversioning system to those kernels does not seem wise, as the overhead added by the training phase can be too high.

On the contrary, the rest of the actors cannot be targeted by Apollo because it is not able to build a prediction model, i.e., it cannot predict the memory accesses and, thus, no transformation is ever tried. Among all the actors presenting this behavior, the most relevant one is *knnSearch*: as mentioned before, this actor is one of the two most time-consuming actors, accumulating near a 50 % of the total execution time, and, thus, it is one of the actors most in need of being parallelized. However, the fact that this actor cannot be targeted by Apollo could be foreseen: it is in charge of looking for the K nearest neighbors of each pixel, i.e., it has to search those neighbors within the window described before and store their position. As their position cannot be possibly known beforehand, their memory addresses are also unknown and, more importantly, cannot be predicted. As a result, this actor can be targeted by PREESM, but not by Apollo.

Once the characterization of the real use-case is completed, the rest of this section is devoted to analyze the performance results obtained with the aforementioned configurations in both architectures. To that end, Table 8-24 gathers the execution times measured for PREESMPar, ApolloPar and CombPar, repeats the sequential times and provides the speedups of each tool when compared to this sequential time, which is considered the baseline. As shown in Equation 8-6, the speedups are computed by comparing the operational time against the sequential one –in other words, supposing an ideal situation in which there are so many iterations of the dataflow graph that the overhead added by the training phase can be considered negligible.

$$Spdp(\text{PREESMPar}) = \frac{\text{Time}}{\text{Seq}} \quad Spdp(\text{ApolloPar/CombPar}) = \frac{\text{Oper}}{\text{Seq}} \quad (8-6)$$

Table 8-24: Execution times measured (in seconds) and speedups achieved (in times) for the dataflow implementation of the real use-case.

Program		Seq	PREESMPar		ApolloPar			CombPar		
			Time	Spdp	Train	Oper	Spdp	Train	Oper	Spdp
Brain-1	Namaka	8.008	3.241	2.5 ×	24.284	6.681	1.2 ×	10.981	2.461	3.3 ×
	Magerit	10.695	3.224	3.3 ×	20.442	7.294	1.4 ×	7.663	3.002	3.6 ×
Brain-2	Namaka	6.757	2.514	2.7 ×	20.998	5.667	1.2 ×	8.368	2.136	3.2 ×
	Magerit	8.976	2.537	3.5 ×	17.480	6.227	1.4 ×	7.590	2.404	3.7 ×
Brain-3	Namaka	4.086	1.589	2.6 ×	13.709	3.229	1.3 ×	8.075	1.361	3.0 ×
	Magerit	5.460	1.550	3.5 ×	11.323	3.603	1.5 ×	7.967	1.410	3.9 ×

To simplify Table 8-24, in this case the iterations of the dataflow graph needed to start having a speedup –*InvC* from the rest of the tables of this chapter– are not provided, since these numbers were analyzed in depth in Section 8.2 both for polyhedral and dataflow domains, so repeating this study here seems redundant.

The same rationale applies to the study of the selected transformation: since there are many actors being targeted by Apollo in the application, to provide that study the transformation chosen for each one should be analyzed; in turn, this means, in the best-case scenario –i.e., *Seq*– providing 5 transformations, and in the worst-case scenario, providing more than 60 –one per instance of actor being targeted– per image.

It should be also noted that, to make Apollo work in the *CombPar* configuration, the number of iterations required by the instrumentation phase needs to be reduced. As described in Section 4.1, Apollo executes an instrumentation phase at the beginning of the processing of each loop kernel; this instrumentation phase is composed by a small set of consecutive iterations of the outermost loop, which are executed sequentially so as to retrieve certain information needed to build the prediction model. By default, this number is set to 16 within Apollo; however, as shown in Table 8-22, each instance of *ImageCentering* and *Covariance* actors processes slices of only 8 bands of the original image, which in turn supposes the outermost loop bound for these kernels to be 8. Hence, if the size of the instrumentation chunk is not reduced, Apollo is not going to be able to perform any optimization in these kernels. Fortunately, this number can be easily modified by the user, so it has been set to 4 to grant Apollo enough time to gather the instrumentation results, build the prediction model and test a transformation.

From the analysis of the results gathered in Table 8-24, it can be inferred that *CombPar* is the configuration achieving the largest speedups regardless of the input image and the target architecture, which proves correct the hypothesis upon which this research work has been built. This configuration is successful in improving the performance results provided by *PREESMPar* in both architectures. On the contrary, when comparing *ApolloPar* and *PREESMPar*, it can be seen that the latter provides better results than the former. The rationale behind this behavior is easily justifiable: *PREESMPar* is able to parallelize both *KnnSarch* and *Covariance*, while Apollo can only attack the latter. This is also the reason why *CombPar* provides better results: it takes advantage of the parallelization possibilities of *PREESM*, letting it parallelize actors that cannot be targeted by Apollo, as *KnnSearch*, and improving the results obtained by *PREESM* in those actors that can be targeted by Apollo, as *Covariance*.

To conclude this study, although the training times have already been analyzed in detail in Section 8.2, there is a last remark that should be addressed. As can be seen in Table 8-24, the training times associated to Namaka are always higher than those associated to Magerit, even though in the operational phase this tendency is inverted. To illustrate this behavior, Figure 8-20 shows the execution time associated to each invocation of the training phase for each image and architecture, where a different transformation from the set is being tested. To simplify the figure, only the times associated to *ApolloPar* are provided as an example. As can be observed, the execution times associated to each invocation are similar for both architectures, except for three cases: transformations 1, 2 and 13. From the figure, it is very clear that these transformations require a much longer time to be tested when compared to the rest of the transformations; additionally, it can also be seen that, in those three cases, the time required by Namaka is considerably larger than that required by Magerit, which

is why, in average, Magerit spends less time in the training phase than Namaka. As a reminder, two aspects must be taken into consideration:

- As the execution time is measured for the whole application and there are several actors invoking the multiversioning system, this execution time gathers all the training times of the actors being targeted by Apollo. Hence, it is possible that the mentioned transformations are taking a long time for only some of the actors, but not as much for the others.
- If the execution time in the training phase is too large, as happens here with transformations 1, 2 and 13, it does not necessarily mean that the transformations are performing badly, since it can also happen that Pluto is requiring an excessively large amount of time to generate the transformation, but the resulting code is very fast, which was the reason why an infinite timeout was selected in the first place.

Figure 8-20: Detail of the execution time measured for each invocation of the training phase for *ApolloPar* in Namaka and Magerit.

Finally, Table 8-25 and Figure 8-21 gather a comparison with the state of the art. As stated at the beginning of this section, the dataflow implementation is based on the implementations developed in the context of HELICoiD. In that line, [Lazcano’19b] presents a thorough analysis of those implementations, providing a comparison of the performance results obtained in different HPC platforms, namely the many-core architecture used in HELICoiD and two GPUs (RTX 2080 and Jetson TX2).

On the one hand, Table 8-25 presents an overview of the main features of each architecture, adding also Namaka, as it is the one providing the fastest results

Table 8-25: Characteristics of the HPC platforms used in [Lazcano'19b].

Platform	#PEs	Frequency (GHz)	Memory (GB)	CPU
RTX 2080	2994	1.71	8	Intel i9 9900X
Jetson TX2	256	1.3	8	ARM Cortex-A57
MPPA-256-N	256	0.55	0.032	Intel i7 3820
Namaka	16	3.4	32	AMD Ryzen™Threadripper™1950X

in Table 8-24; on the other hand, Figure 8-21 provides the comparison of the execution times measured for the use-case described here in each of the architectures aforementioned. The times associated to RTX 2080, Jetson TX2 and MPPA-256-N have been directly extracted from [Lazcano'19b], while the times associated to Namaka correspond with *CombPar-Oper* from Table 8-24.

Figure 8-21: Comparison of the execution times of *CombPar-Oper* in Namaka and the implementations presented in [Lazcano'19b].

As can be seen, the methodology proposed in this document produces faster results than those achieved with the MPPA-256-N, and very similar ones to those obtained with the Jetson TX2. Hence, considering the number of PEs of each platform, it can be concluded that this methodology can compete with implementations in which the parallelization has been manually exploited, as in [Lazcano'19b].

8.4 Discussion

This chapter has thoroughly evaluated and validated the methodology proposed in this document to further increase the parallelization possibilities of dataflow frameworks by providing them with automatic parallelization capabilities through polyhedral transformations. To do so, this evaluation has been divided into two separate studies: first, a functional validation has been carried out by means of two different set of benchmarks, each one coming from one of the specific domains this methodology combines; secondly, a use-case validation has been addressed, in which a real-life application has been selected and implemented following this methodology so as to check if the expected advantages are such or not.

On the one hand, regarding the functional validation, this process was in turn divided into two phases, one per associated domain: first, the functional validation was carried out for the polyhedral domain, using PolyBench and ApolloBench benchmark suites to validate the changes and extensions added within Apollo; likewise, a similar process was addressed for the selected dataflow benchmarks, which were three applications traditionally used by PREESM developers to test new functionalities.

- The evaluation of the proposed methodology through the polyhedral benchmarks showed that the multiversioning system included in Apollo's runtime phase always provides better results than those obtained with the original version of Apollo for applications simulating a dataflow behavior. As for Clang, it was shown that the multiversioning system provides better results for larger sizes, but that it also obtained very interesting results for smaller ones, where Apollo was not expected to obtain any speedup. Yet, for all cases, these improvements came at the cost of adding an overhead in the training phase that may not be negligible. Finally, it was proven that, depending on the environment conditions –architecture, problem size, kernel and timeout–, the transformation selected as the best performing one varied, and that its impact on the performance was considerable.
- The evaluation of the proposed methodology through the dataflow benchmarks was divided again into two studies: first, the extensions added within PREESM were validated through three applications widely used by PREESM developers to test new functionalities –Sobel, Stereo and Matmul–; secondly, the changes added within SPiDER –PREESM runtime manager– were also validated with one of the aforementioned applications. Apart from validating the new functionalities of each tool, one of the main objectives was to study the advantages and disadvantages of granting the parallelization potential to one domain or the other. In this context, the first study showed that these applications provide representative examples for this objective, since Sobel achieves better results when PREESM is in charge, Stereo reached larger speedups with Apollo, and Matmul proved to be much faster when both

parallelization capabilities were combined. With respect to the second study, it showed that the multiversioning system successfully detects changes at runtime and reconfigures itself, providing similar performance results than those obtained with PREESM for Matmul application.

On the other hand, the application selected for the real use-case validation was the supervised classification stage of the custom algorithm developed within HELICoiD, which aimed at processing hyperspectral images capturing human brain surfaces to detect brain tumors during surgery. The objective of this study was to use a real-life application to show the advantages of using the proposed methodology to model, deploy and automatically optimize applications. The conclusions of this study were twofold: first, the obtained results showed that, indeed, the largest speedups were achieved when using the *CombPar* configuration, which is the one combining the parallelization capabilities of PREESM and Apollo; secondly, a comparison with a state-of-the-art study [Lazcano'19b] demonstrated this methodology to achieve competitive results in terms of performance when compared to implementations deployed onto HPC platforms where the parallelization is manually exploited.

Hence, it has been proven that using this methodology helps in conjugating the parallelization capabilities of both worlds, taking advantage of each one separately in those parts where one fails to provide performance improvements, and joining both potentials in those parts that can be targeted by both domains.

9

CONCLUSIONS

During this document, the main ideas of this PhD have been stated, described, evaluated and validated. Once all this information has been presented, this chapter closes the document by extracting and summarizing the main conclusions and contributions of this work; in addition, a list of the future research lines that have appeared as a consequence of this work is also provided.

To do so, Section 9.1 outlines the most relevant conclusions extracted from the work presented in this document and developed during this PhD, while Section 9.2 summarizes the main contributions of this thesis. Similarly, Section 9.3 introduces the academic production developed in the context of this PhD, including scientific publications, collaborations, participation in research projects, open-access products and grants and awards obtained during the production of this work. Finally, Section 9.4 closes this document by listing the research lines born as a result of this work, either to improve the work already developed or to extend it to other application fields.

9.1 Conclusions

As described in Section 1.1, nowadays, the increase in (i) the volumes of information to process, (ii) the algorithms complexity, and (iii) the performance requirements has contributed to the need of parallel processing and the usage of multicore architectures. These different challenges have helped in widening the gap between platform usage complexity and application development productivity. In the recent years, many methodologies have been developed as a solution to bridge this productivity gap, such as Y-chart design strategy, dataflow-based design or automatic parallelization techniques. Y-chart design isolates the application from the architecture, modeling each one separately and merging them afterwards; dataflow-based design proposes modeling applications following dataflow MoCs, which are specifically well-suited to expose their structural parallelism; automatic parallelization techniques aim at parallelizing applications without developers having to manually handle it.

The combination of the first two yielded a methodology in which a dataflow-based application is modeled as a set of tasks –or *actors*– exchanging information –or *tokens*– that can be automatically deployed onto multicore architectures, mapping and scheduling these tasks so as to use all the available resources of the target architecture. However, this process is still arduous from the developer’s point of view: devising an

efficient mapping and scheduling of these tasks requires many iterations, since the computational load of tasks may completely differ. As these tasks are the functional units of these models, thus they are considered as black boxes –primitives of the system–, so up until now there is an open issue to solve if the computational load of the tasks is not equally balanced.

To tackle this problem, the main objective of this work –labeled as **[OBJ-P]** in Section 1.2– is to devise an improved methodology to further optimize applications in terms of performance by combining these three methodologies with the objective of being used with a wide range of applications, simplifying the process of parallelizing and deploying applications onto multicore platforms. This document hence proposes the combination of a Y-chart dataflow-based framework, as PREESM, with an automatic parallelization and optimization tool, like Apollo, which applies aggressive transformations at runtime to increase applications performance **[OBJ-S1]**.

The combination of these two frameworks opens, in turn, a wide range of opportunities, since the characteristics intrinsic to dataflow MoCs can be fed to Apollo to (i) make it compliant with dataflow-based applications, and (ii) explore further optimization possibilities appearing from these characteristics, and which were out of reach until now. Consequently, Apollo has been extended to cope with these two features **[OBJ-S1.1]**, doting its runtime manager with a multiversioning system capable of evaluating different optimizing transformations and choosing the best one depending on their context, as described in Chapter 5.

Likewise, once the automatic optimization tool has been modified and extended to include these new functionalities, it needs to be embedded within the dataflow framework to combine their parallelization potentials. Consequently, Apollo has been embedded within both PREESM and SPiDER **[OBJ-S1.2]**, studying how to balance the parallelization potential of both frameworks to make the most of them, as detailed in Chapter 6. By doing so, further optimizations can be performed inside the actors composing the applications to exploit not only inter-actor parallelism –i.e., data parallelism–, but also intra-actor optimization opportunities.

As the objective is to apply this methodology to a wide range of applications, a heterogeneous benchmark suite has been devised **[OBJ-S2]** to validate it with applications coming from different domains, as established in Section 7.1. To do so, this benchmark suite has been defined as a combination of three benchmark suites, two coming from the polyhedral domain –PolyBench and ApolloBench– and one from the dataflow world –composed of three applications traditionally used to test new functionalities within PREESM.

As detailed in Subsection 8.2.1, the evaluation of the proposed methodology with the polyhedral benchmarks showed that, as expected, the results obtained with the extended version of Apollo outperform by far those obtained with the original one for applications simulating a dataflow-based behavior, providing specially good results for smaller problem sizes. With respect to Clang, the results show that the throughput is also improved, specially for larger problem sizes, where maximum speedups of $161\times$ and $59\times$ are achieved for ApolloBench and PolyBench, respectively. The evaluation

of these benchmarks also produced interesting results related to the transformation selected for each case, since it was proved that the transformation has a direct impact on the execution time, meaning that, depending on the benchmark, problem size or target architecture, a transformation may provide very good or very bad results, hence showing the benefits of a multiversioning system. Additionally, it was also proved that the usage of such a system comes at the cost of adding a non-negligible overhead, but it can be canceled out thanks to the iterative nature of dataflow applications.

Additionally, as presented in Subsection 8.2.2, the evaluation of the proposed methodology with the dataflow benchmarks was devoted to assess the advantages and disadvantages of granting the parallelization possibilities to one tool or the other. The three applications selected turned out to be a good representation of this problematic: Sobel obtained better results when PREESM dealt with the parallelization, providing speedups near $9\times$ when compared to Clang, while Apollo only reached $4\times$ and the combination less than $6\times$; Stereo presented a speedup only when Apollo was in charge –around $5\times$ –; finally, Matmul proved to be much faster when the optimizations came from both frameworks, since the speedups reached were almost $100\times$ for the combination, $14\times$ for PREESM and $75\times$ for Apollo. Matmul application was also used to functionally validate the modifications performed within SPiDER, showing that the proposed methodology is successful in detecting runtime changes and reconfiguring the behavior of the multiversioning system accordingly, reaching similar speedups than those obtained statically –almost $100\times$.

Once the proposed methodology has been functionally validated through the benchmark suite specifically composed for this task, the application selected as the real use-case –Section 7.2– has been implemented using this methodology so as to characterize its applicability through a real-life application [OBJ-S3]. The use-case has been extracted from a custom algorithm developed in the context of HELICoiD project to detect human brain tumors in intraoperative real-time during surgical procedures: specifically, this use-case implements a supervised classification, which is the stage of the complete algorithm demanding the most computational resources. As stated in Section 8.3, the results obtained proved that, as expected, the largest speedups were achieved when the optimization potential of both tools is combined, reaching speedups near $4\times$ when compared to Clang. The results obtained with this methodology have proven to be competitive with those extracted from the state of the art, showing similar performance with considerably less PEs and using automatic parallelization techniques, instead of implementations manually parallelized.

As a conclusion, the results obtained both with the benchmark suites and the real use-case application prove that the methodology proposed in this document is successful in conjugating the parallelization capabilities of dataflow-based approaches with the optimization potential of automatic parallelization techniques, which is achieved by either taking advantage of each world separately in those cases where one fails to provide performance improvements, or exploiting both potentials in those cases that can be targeted by both domains.

9.2 Contributions

After extracting the main conclusions of this work and verifying the fulfillment of the objectives established for this PhD, this section summarizes the original contributions presented in this document. These contributions are going to be highlighted following the order of the manuscript: first, Subsection 9.2.1 presents the contributions associated to extending Apollo so as to make it dataflow-compliant and take advantage of the characteristics intrinsic to these type of applications; afterwards, Subsection 9.2.2 highlights the original contributions of this work to the extension of PiSDF MoC through polyhedral transformations, considering the integration of Apollo with PREESM and SPiDER; finally, Subsection 9.2.3 analyzes the contributions related to the functional validation of the methodology presented in this document, while Subsection 9.2.4 outlines those associated to the use-case selected to characterize the applicability of such a methodology to a real-life problem.

9.2.1 Dataflow-based runtime polyhedral optimizations

As stated throughout this document, the main objective of this PhD is to devise a methodology that helps in further optimizing applications by combining Y-chart dataflow-based design parallelization capabilities with the optimization potential offered by the polyhedral model, as depicted in Figure 9-1. To do so, the first step is to take advantage of the specific characteristics associated to dataflow applications to adapt the polyhedral-based optimizations to this behavior, getting the most out of their iterative nature while making these optimizations dataflow-compliant.

To that end, one of the original contributions of this PhD is to feed these dataflow features to a framework as Apollo, which already applies polyhedral transformations at runtime, so as to apply them inside dataflow actors; as they are considered functional units, no optimization is ever applied. In this regard, a runtime multiversioning system that applies optimization techniques such as memoization and code specialization has been embedded within Apollo's runtime manager as detailed in Chapter 5. This contribution can be divided into several individual ones as follows:

- **Multiversioning:** As described in Section 5.1, the envisioned system tests different optimizing transformations with the objective of finding the most suitable one for each case, which may vary depending on many features, such as application, target architecture or volume of data to be processed. Additionally, the envisioned system includes a specific mechanism to evaluate these versions and accurately choose the best one. This mechanism consists of computing at runtime Ehrhart polynomial –Section 5.2–, which provides the total number of iterations of a loop, so as to consider the computational load of each transformation when assessing its performance.

Figure 9-1: Overview of the proposed methodology (based on Figure 1-2).

- **Memoization:** Once the best version has been chosen, this feature allows storing this version in non-volatile memory –Section 5.3–. In this way, if the same loop is ever launched again with the same environment conditions, this transformation can directly be retrieved, without having to generate and evaluate it again.
- **Code specialization:** Best versions deeply depend on environment features as the kernel being targeted, the size of data this kernel processes or characteristics of the platform where it is being executed. Additionally, dynamic contexts present parameters that may change at runtime, and that can affect the kernel in several ways –e.g., varying the loop bounds–. Thanks to code specialization –Section 5.5–, the envisioned system can actively consider them, making the best versions depend upon the specific values of those parameters, storing one specialized best version per value of parameters for each targeted kernel.

As a result, Apollo’s runtime manager has been provided with a multiversioning system that combines all the aforementioned features. This system is divided into two successive phases: training phase and operational phase. In the training phase, a set of transformations are evaluated, so as to find the best performing one for each case; once this best version has been selected, the operational phase begins, in which the best version is directly retrieved whenever the same kernel with the same parameters is launched again. This system has been designed to be flexible; as a result, both the specific optimizations to test and the number of transformations to evaluate before launching the operational phase can be easily configured by users.

9.2.2 PiSDF MoC extension through polyhedral transformations

Once Apollo is modified to comply with dataflow-based applications and get the most out of their intrinsic features, the last step to implement this methodology is to embed Apollo into a dataflow design framework as PREESM –Chapter 6–. As shown in Figure 6-1 –repeated in Figure 9-2–, the new functionalities appearing due to this integration can be divided into three concerns, considering their area of application:

Figure 9-2: Overview of the specific tasks envisioned to embed the polyhedral domain in the Y-chart, dataflow-based design methodology (repeated from page 90).

- **Polyhedral-dataflow interface ❶:** One of the main steps towards fully integrating dataflow and polyhedral domains is to set up communication channels between both domains, so the required information can be passed through one to the other, and vice versa. In this regard, the original contribution of this thesis is the creation of an API to set up these channels, but also to provide PREESM with the necessary mechanisms to configure Apollo –Section 6.1.
- **Dataflow preparation ❷:** This category refers to the changes added within Apollo to prepare it for dataflow applications –Section 6.2–. As previously said, one of the benefits of the proposed methodology is the possibility to balance the parallelization potential of PREESM with the optimization capabilities of Apollo, prioritizing one in particular or joining both potentials depending on the

obtained results. Hence, both tools need to be adapted so their parallelization potentials do not interfere. The original contributions in this regard are:

1. The creation of different working modes within Apollo to suppress any kind of multithreaded activity when this tool is configured just as an optimizer.
 2. The adaptation of Apollo so as to make it usable in multithreaded environments, since this tool was originally devised to be standalone and, therefore, some concurrency issues could appeared either in Apollo or in any of the tools it relies upon, as Pluto.
 3. The creation in PREESM of an identifier of the kernel being targeted so as to feed it to Apollo, so different kernels within the same application can be targeted by the latter. As this identifier must uniquely identify the kernel under study, it must contain (i) an identifier of the loop nest, which is already provided by Apollo; (ii) an identifier of the actor embedding the kernel under study; and (iii) the combination of the specific values of the dynamic parameters when the kernel is being executed, so as to associate its own context to each value of these parameters and allow the reconfiguration of the system when these parameters change.
- **Polyhedral preparation ⑨:** This category refers to the changes needed within a dataflow application for it to be optimized with Apollo –Section 6.3–. These changes are grouped into three categories, following a top-down approach:
 1. **Framework-level changes:** As already described, Apollo needs to compile the application with a specific compiling toolchain so certain information needed by its runtime is extracted; consequently, the main contribution in this aspect has been changing the compiling toolchain of both PREESM and SPiDER to use the one required by Apollo when users express their wish to use the proposed methodology.
 2. **Application-level changes:** As stated at the beginning of this section, one of the objectives of the designed API is to set up a two-way communication between both domains. Yet, to simplify the application of the proposed methodology as much as possible, the usage of this API has been automated within the code generation process of PREESM and SPiDER, requiring from the user only the activation of Apollo. As a result, the calling to the functions of this API has been embedded within both tools, making it depend on the activation of Apollo from users' side.
 3. **Actor-level changes:** The configuration of each actor is also automatically sent to Apollo through the code generation process of PREESM and SPiDER. Once this is done, the only aspect remaining is to include Apollo's dedicated pragma surrounding those kernels the user wishes to optimize with Apollo. Since this process is the only one that must be performed manually by the user, a tutorial has been uploaded to PREESM website to guide developers through this last step.

9.2.3 Heterogeneous benchmarking

Once the methodology has been designed and implemented, it must be validated. To that end, one of the contributions of this thesis is the creation of a heterogeneous set of applications coming from both domains, polyhedral and dataflow, to (i) have enough representation of all the characteristics intrinsic to both worlds; and (ii) validate the methodology with applications traditionally used in both domains, thus simplifying the comparison with other techniques of the state of the art, as depicted in Chapter 7.

- **Polyhedral domain:** In this domain, PolyBench clearly stands out among the rest of the benchmark suites, since it is the reference one for validating tools based on the polyhedral model. Yet, as Apollo extended the polyhedral model to runtime, its developers built their own benchmark suite, ApolloBench, to evaluate some features that were out of the scope of PolyBench. Consequently, it has been decided to combine both benchmark suites to cover all possibilities, adapting both benchmarks so as to simulate a dataflow behavior –i.e., iterating the same kernel many times–. Between both suites, more than 25 applications are evaluated, and the obtained results –Subsubsection 8.2.1.3– prove that (i) the extended version of Apollo always outperforms the original one when targeting dataflow-like applications; (ii) speedups near $161\times$ and $59\times$ are obtained for the two target architectures when compared to Clang; and (iii) the best performing version is deeply correlated to the kernel under evaluation, the size of data it handles, the timeout configured for Apollo or the characteristics of the target platform, as shown in Subsubsection 8.2.1.2.
- **Dataflow domain:** On the contrary, dataflow domain does not present a uniform benchmark suite widely used in the domain. Since dataflow MoCs may present extremely different characteristics, it is not easy to devise a uniform benchmark for all of them; consequently, the most usual praxis is for each developer community to use their own set of applications with which new functionalities are validated. For that reason, three of the applications PREESM developers use the most to validate new functionalities have been chosen to complete the benchmark suite. The obtained results –Subsection 8.2.2– prove these applications to be a good representation, since they are successful in providing examples where handing over the parallelization possibilities to one framework or the other results in very different performances: one is faster when PREESM parallelizes, another maximizes its throughput when Apollo is in charge, and the third one reaches much larger speedups when both of them join their capabilities. The latter is used additionally to validate the changes performed within SPiDER, showing speedups of almost $100\times$ when comparing to Clang.

Through this heterogeneous set of applications, the methodology proposed in this document is thoroughly validated, studying in detail all the possibilities such a methodology can offer.

9.2.4 HSI cancer detection

Although the functional validation carried out with the heterogeneous benchmark suite specifically built for that purpose highlighted all the benefits this methodology can provide, the last step before closing this document is to apply this methodology to a real-life problem. The real-life problem that has been chosen is the application of HSI technology to human brain cancer detection to assist specialists in locating tumor boundaries during surgical procedures. This application, developed in the context of HELICoiD European project, fulfills the requirements for which this methodology has been envisioned, since it involves the implementation of highly demanding algorithms and their deployment onto a multicore architecture. Specifically, the part conforming the use-case of this study is a supervised classification procedure –Section 7.2–, which has been selected because it is the part demanding most computational resources, to the point that, in the implementation devised during the project, this part was offloaded to a hardware accelerator, namely to an MPPA-256-N. Specifically, the original contributions of this PhD to this regard are the following:

- **Implementation by hand:** As already mentioned, in the context of HELICoiD project an implementation of the supervised classification stage was developed and deployed onto an MPPA-256-N. This hardware accelerator gathered 256 PEs and was chosen at the beginning of the project due to its low power consumption, but the parallelization of the algorithm had to be manually addressed. This resulted in an arduous process, which in turn generated many studies until an efficient implementation was devised.
- **Dataflow implementation:** Dataflow MoCs are specifically well-suited to expose the structural parallelism of an application, so a new implementation of the use-case was devised and presented in Subsection 8.3.1, now modeled in PREESM according to PiSDF MoC, as it presents many interesting features:
 - The devised implementation is *parameterized*, which means that many aspects can be easily configured by the user through parameters, as the size of the images to be processed or the level of parallelism to apply.
 - This implementation is also *modular*, as it has been devised following a top-down approach. This characteristic implies that the process of changing the algorithms composing the application, or any part within them, is straightforward: as long as the inputs and outputs of actors are not modified, the implementation of each actor can be directly substituted.
 - Finally, the devised implementation is *portable*, since one of the main advantages of PREESM framework is that it supports a wide range of architectures onto which the application can be directly deployed, requiring no extra effort from developers side.
- **Polyhedral+dataflow implementation:** Finally, the dataflow implementation has been adapted so as to use the methodology proposed in this document

to exploit the optimization possibilities offered by the polyhedral model while maintaining also the parallelization potential of dataflow-based approaches. The results obtained for the use-case –Subsection 8.3.3– proved speedups near $4\times$, which outperformed the results provided by the dataflow implementation and, more importantly, outperformed also the results achieved with the implementation devised for HELICoiD project. This result is especially relevant, since MPPA-256-N gathers many more PEs than the architectures used in this work –256 versus 16 and 20, respectively– and required considerably less effort from developers, actively contributing to reduce the time invested in devising an implementation trying to make an efficient usage of the available resources.

9.3 Impact of the thesis

In this section, the academic production carried out during this PhD is summarized. To that end, this section is divided into six categories: (i) scientific publications, (ii) participation in national and international research projects, (iii) collaborations with other institutions, (iv) open-access products, (v) grants and (vi) awards received.

9.3.1 Publications and dissemination

During the development of this PhD, several works have been published highlighting some of the contributions presented in this work. Additionally, scientific dissemination has been fulfilled by means of posters and demonstrators.

9.3.1.1 Journal publications

The author of this work has authored or co-authored 11 journal articles related with the work presented in this manuscript, which are summarized below:

[Lazcano'19b] R. Lazcano, D. Madroñal, G. Florimbi, J. Sancho, S. Sanchez, R. Leon, H. Fabelo, S. Ortega, E. Torti, R. Salvador, et al., “Parallel Implementations Assessment of a Spatial-Spectral Classifier for Hyperspectral Clinical Applications”, in *IEEE Access*, vol. 7, pp. 152 316–152 333, 2019. **(Q1)**

This paper presents a comparative study evaluating the performance and energy consumption of the implementation of a part of the algorithm developed within HELICoiD project to process medical hyperspectral images in real-time when executed in different hardware accelerators, such as many-cores and GPUs. The implementations presented here are taken as a basis for the real use-case of this thesis –Chapter 7 and Chapter 8.

- [Madronal'19] D. Madronal, F. Arrestier, J. Sancho, A. Morvan, R. Lazcano, K. Desnos, R. Salvador, D. Menard, E. Juarez, C. Sanz, "PAPIFY: Automatic Instrumentation and Monitoring of Dynamic Dataflow Applications Based on PAPI", in *IEEE Access*, vol. 7, pp. 111 801–111 812, 2019. **(Q1)**

This paper presents PAPIFY toolbox, describing both its functioning and its integration with PREESM and SPiDER. This tool is used in Chapter 8.

- [Fabelo'19b] H. Fabelo, S. Ortega, A. Szolna, D. Bulters, J. F. Piñeiro, S. Kabwama, A. JO'Shanahan, H. Bulstrode, S. Bisshopp, B. R. Kiran, et al., "In-vivo hyperspectral human brain image database for brain cancer detection", in *IEEE Access*, vol. 7, pp. 39 098–39 116, 2019. **(Q1)**

This paper introduces the human brain HSI database generated as a result of the work developed within HELICoiD project. This database is used in Chapter 8.

- [Lazcano'19a] R. Lazcano, D. Madroñal, H. Fabelo, S. Ortega, R. Salvador, G. Callico, E. Juarez, C. Sanz, "Adaptation of an iterative PCA to a manycore architecture for hyperspectral image processing", in *Journal of Signal Processing Systems*, vol. 91, no. 7, pp. 759–771, 2019. **(Q4)**

This paper presents a study of an iterative implementation of PCA algorithm, known as Non-Linear Iterative Partial Least Squares (NIPALS), onto an MPPA-256-N many-core architecture, which is evaluated as an alternative to standard implementations of PCA.

- [Florimbi'18] G. Florimbi, H. Fabelo, E. Torti, R. Lazcano, D. Madroñal, S. Ortega, R. Salvador, F. Leporati, G. Danese, A. Báez-Quevedo, et al., "Accelerating the K-nearest neighbors filtering algorithm to optimize the real-time classification of human brain tumor in hyperspectral images", in *Sensors*, vol. 18, no. 7, p. 2314, 2018. **(Q1)**

This paper presents a study of the KNN algorithm, which is a part of the algorithm devised within HELICoiD project to process the images captured at the operating theater, to devise an efficient implementation that optimizes and parallelizes this algorithm by exploiting GPU technology.

- [Martel'18] E. Martel, R. Lazcano, J. López, D. Madroñal, R. Salvador, S. López, E. Juarez, R. Guerra, C. Sanz, R. Sarmiento, "Implementation of the principal component analysis onto high-performance computer facilities for hyperspectral dimensionality reduction: Results and comparisons", in *Remote Sensing*, vol. 10, no. 6, p. 864, 2018. **(Q1)**

This paper presents a comparative study of the implementation of PCA algorithm, which is a part of the algorithm devised within HELICoiD project to process the images captured at the operating theater, onto two different high performance devices, namely a many-core and a GPU, and using hyperspectral images widely used in the community of remote sensing.

- [Fabelo'18b] H. Fabelo, S. Ortega, D. Ravi, B. R. Kiran, C. Sosa, D. Bulters, G. M. Callicó, H. Bulstrode, A. Szolna, J. F. Piñeiro, et al., "Spatio-spectral classification of hyperspectral images for brain cancer detection during surgical operations", in *PLoS One*, vol. 13, no. 3, 2018. **(Q2)**

This paper outlines the algorithm devised for processing the medical hyperspectral images obtained in HELICoiD project. A part of this algorithm is used in this work as the real use-case, described in Chapter 7 and evaluated in Chapter 8.

- [Fabelo'18a] H. Fabelo, S. Ortega, R. Lazcano, D. Madroñal, G. M. Callicó, E. Juárez, R. Salvador, D. Bulters, H. Bulstrode, A. Szolna, et al., "An intraoperative visualization system using hyperspectral imaging to aid in brain tumor delineation", in *Sensors*, vol. 18, no. 2, p. 430, 2018. **(Q1)**

This paper presents the demonstrator developed within the context of HELICoiD project to help surgeons in delineating the boundaries of brain tumors during surgical procedures. This demonstrator has been used to obtain the images used as input of the use-case presented in Chapter 7 and assessed in Chapter 8.

- [Madroñal'17b] D. Madroñal, R. Lazcano, R. Salvador, H. Fabelo, S. Ortega, G. M. Callicó, E. Juárez, C. Sanz, "SVM-based real-time hyperspectral image classifier on a manycore architecture", in *Journal of Systems Architecture*, vol. 80, pp. 30–40, 2017. **(Q3)**

This paper presents an analysis of the design space of an SVM classifier implemented onto an MPPA-256-N many-core architecture, which is also a part of the algorithm devised within HELICoiD project to process the images captured at the operating theater.

- [Lazcano'17c] R. Lazcano, D. Madroñal, R. Salvador, K. Desnos, M. Pelcat, R. Guerra, H. Fabelo, S. Ortega, S. López, G. M. Callicó, et al., "Porting a PCA-based hyperspectral image dimensionality reduction algorithm for brain cancer detection on a manycore architecture", in *Journal of Systems Architecture*, vol. 77, pp. 101–111, 2017. **(Q3)**

This paper presents a study of the parallelism exploitation of the different stages of PCA algorithm and their adaptation to an MPPA-256-N many-core architecture with the objective of fulfilling the performance requirements established by HELICoiD project.

- [Ortega'17] S. Ortega, H. Fabelo, R. Camacho, M. Plaza, G. Callico, R. Lazcano, D. Madroñal, R. Salvador, E. Juárez, R. Sarmiento, "P03. 18 Detection of human brain cancer in pathological slides using hyperspectral images", in *Neuro-oncology*, vol. 19, no. Suppl 3, p. iii37, 2017. **(Q1)**

This paper presents a brief overview of the research carried out within HELICoiD project to delineate brain tumor boundaries during neurosurgical procedures.

9.3.1.2 Conference publications

Regarding conference publications, the following contributions related to the work presented in this document have been authored or co-authored:

- [Lazcano'20] R. Lazcano, D. Madroñal, E. Juárez, P. Clauss, "Runtime multi-versioning and specialization inside a memoized speculative loop optimizer", in *Proceedings of the 29th International Conference on Compiler Construction*, pp. 96–107, 2020.

This paper outlines the main modifications carried out within Apollo so as to turn it into a runtime multiversioning system including memoization and code specialization. These modifications are described in detail in Chapter 5.

- [Aragón'19] J. S. Aragón, S. S. Ramírez, R. L. López, D. M. Quintín, R. S. Perea, E. J. Martínez, C. S. Álvaro, "Characterizing Hyperspectral Data Layouts: Performance and Energy Efficiency in Embedded GPUs for PCA-based Dimensionality Reduction", in *2019 XXXIV Conference on Design of Circuits and Integrated Systems (DCIS)*, pp. 1–6, IEEE, 2019.

This paper reviews the different data layouts for hyperspectral images in terms of performance and energy efficiency for an embedded GPU when using dimensionality reduction algorithms, such as PCA.

- [Palumbo'19b] F. Palumbo, T. Fanni, C. Sau, A. Rodríguez, D. Madroñal, K. Desnos, A. Morvan, M. Pelcat, C. Rubattu, R. Lazcano, et al., "Hardware/Software Self-adaptation in CPS: The CERBERO Project Approach", in *International Conference on Embedded Computer Systems*, pp. 416–428, Springer, 2019.

This paper provides an overview of Cross-layer model-based framework for multi-objective design of Reconfigurable systems in uncertain hybrid environments (CERBERO) project and, more specifically, of its efficient support for runtime hardware/software self-adaptivity, describing the infrastructure of the adaptation loop for Cyber-Physical System (CPS) and presenting the achieved mid-term results.

- [Madroñal'18] D. Madroñal, A. Morvan, R. Lazcano, R. Salvador, K. Desnos, E. Juárez, C. Sanz, "Automatic instrumentation of dataflow applications using papi", in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pp. 232–235, ACM, 2018.

This paper describes an early work on PAPIFY, outlining the first ideas on how to access to hardware monitors from a high level of abstraction and how to integrate this with a dataflow tool like PREESM.

- [Madroñal'17a] D. Madroñal, R. Lazcano, H. Fabelo, S. Ortega, R. Salvador, G. M. Callicó, E. Juárez, C. Sanz, "Energy consumption characterization of a Massively Parallel Processor Array (MPPA) platform running a hyperspectral SVM

classifier”, in *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1–6, IEEE, 2017.

This paper presents a characterization in terms of energy consumption of an MPPA-256-N many-core architecture through an implementation of an SVM algorithm for classifying hyperspectral images.

- [Lazcano’17b] R. Lazcano, D. Madroñal, H. Fabelo, S. Ortega, R. Salvador, G. M. Callicó, E. Juárez, C. Sanz, “Parallel implementation of an iterative PCA algorithm for hyperspectral images on a manycore platform”, in *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1–6, IEEE, 2017.

This paper surveys an early work on implementing a parallelized version of NIPALS algorithm onto an MPPA-256-N many-core architecture, which can be used as an alternative to PCA for dimensionality reduction purposes.

- [Lazcano’17a] R. Lazcano, D. Madroñal, H. Fabelo, S. Ortega, R. Salvador, G. Callicó, E. Juárez, C. Sanz, “Parallel exploitation of a spatial-spectral classification approach for hyperspectral images on RVC-CAL”, in *High-Performance Computing in Geoscience and Remote Sensing VII*, vol. 10430, p. 104300A, International Society for Optics and Photonics, 2017.

This paper presents an implementation of PCA, SVM and KNN algorithms in a dataflow fashion, using a dataflow language known as RVC-CAL. This paper was awarded the **Best Student Paper Award** of the conference.

- [Domingo’17] R. Domingo, R. Salvador, H. Fabelo, D. Madroñal, S. Ortega, R. Lazcano, E. Juárez, G. Callicó, C. Sanz, “High-level design using Intel FPGA OpenCL: A hyperspectral imaging spatial-spectral classifier”, in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, IEEE, 2017.

This paper describes an implementation of KNN algorithm for Intel Cyclone V System-on-Chip (SoC) using OpenCL, comparing the obtained results with a baseline C implementation.

- [Salvador’17] R. Salvador, S. Ortega, D. Madroñal, H. Fabelo, R. Lazcano, G. Marrero, E. Juárez, R. Sarmiento, C. Sanz, “HELICoiD: Interdisciplinary and collaborative project for real-time brain cancer detection”, in *Proceedings of the Computing Frontiers Conference*, pp. 313–318, ACM, 2017.

This paper provides an overview of HELICoiD project, presenting its main objectives, assessing its scientific and technical challenges and highlighting the main outcomes expected to be obtained by the end of the project.

- [Madroñal’16b] D. Madroñal, R. Lazcano, H. Fabelo, S. Ortega, G. M. Callicó, E. Juárez, C. Sanz, “Hyperspectral image classification using a parallel implementation of the linear SVM on a Massively Parallel Processor Array (MPPA) platform”, in *2016*

Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 154–160, IEEE, 2016.

This paper provides an early work on exploiting the parallelism of an SVM algorithm running on an MPPA-256-N many-core architecture, proving the advantages of using this kind of platform to parallelize complex applications while maintaining the energy consumption within reasonable limits.

- [Lazcano'16b] R. Lazcano, I. Sidrach-Cardona, D. Madroñal, K. Desnos, M. Pelcat, E. Juárez, C. Sanz, “Parallelism exploitation of a PCA algorithm for hyperspectral images using RVC-CAL”, in *High-Performance Computing in Geoscience and Remote Sensing VI*, vol. 10007, p. 100070H, International Society for Optics and Photonics, 2016.

This paper provides an implementation of PCA algorithm using a dataflow-based language, such as RVC-CAL, showing the advantages of using dataflow programming to expose the inner parallelism of complex algorithms.

- [Madroñal'16a] D. Madroñal, H. Fabelo, R. Lazcano, G. Callicó, E. Juárez, C. Sanz, “Parallel implementation of a hyperspectral image linear SVM classifier using RVC-CAL”, in *High-Performance Computing in Geoscience and Remote Sensing VI*, vol. 10007, p. 1000709, International Society for Optics and Photonics, 2016.

This paper presents an RVC-CAL implementation of SVM algorithm, highlighting the advantages of using dataflow programming to expose the inner parallelism of complex applications.

9.3.1.3 Other dissemination channels

In addition to the journal and the conference publications, the work proposed in this PhD has been disseminated through posters and demos in peer-reviewed conferences.

- [Lazcano'16a] R. Lazcano, D. Madroñal, K. Desnos, M. Pelcat, R. Guerra, S. López, E. Juárez, C. Sanz, “Parallelism Exploitation of a Dimensionality Reduction Algorithm Applied to Hyperspectral Images”, in *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2016. **Poster**.

This poster presents an early work studying the parallelism exploitation of PCA algorithm and its adaptation to an MPPA-256-N many-core architecture.

- [Salvador'16] R. Salvador, H. Fabelo, R. Lazcano, S. Ortega, D. Madroñal, G. M. Callicó, E. Juárez, C. Sanz, “HELICoiD tool demonstrator for real-time brain cancer detection”, in *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 237–238, IEEE, 2016. **Demo**.

This demonstrator shows how the combination of HSI and machine learning is a potential solution to precise real-time detection of tumor tissue during neurosurgeries by showing the processing of HSI images on an MPPA-256-N. This demo was awarded the **DASIP 2016 Best Demo Award**.

9.3.2 Research projects

The candidate has actively participated in 3 research projects during the PhD, which have contributed to the development of the work presented in this document.

- **HypErspectraL Imaging Cancer Detection (HELICoiD)**: HELICoiD was funded by the European Commission through the FET-Open call, under the FP7 program of the European Union (FP7-ICT-2013.9.2 (FET Open) Grant agreement ID: 618080). This project was born as a collaboration among two hospitals, three industrial partners and four universities, and its objective was to use HSI to provide surgeons with enough information to distinguish tumor boundaries during human brain surgical procedures. This project ended in 2017, and its contribution to the state of the art was twofold: (i) a demonstrator was built and successfully used in real-time during surgeries; (ii) an HSI in-vivo database was created and published as open-source with the objective of pushing the research on this topic. In this context, the candidate has actively contributed to accelerate the algorithms in the target architecture, so the demonstrator reaches the desired real-time objective. As a result, as described in Section 7.2, part of the algorithm developed in this project has been selected as the real use-case of this PhD.
- **Cross-layer model-based fRamework for multi-oBjective dEsign of Reconfigurable systems in unceRtain hybRid enviroNments (CERBERO)**: CERBERO was also funded by the European Commission, this time through the H2020 call (H2020-ICT-2016-1 Grant agreement ID: 732105), and it was born as a collaboration among five universities and seven industrial partners. It ended in 2019, and its main goal was the development of a design environment for CPS proposing a model-based approach to describe, optimize and analyze the target system, together with an advanced runtime self-adaptation loop to continuously monitor, optimize and reconfigure the target system so as to provide high performance, robustness and reactivity. The candidate has been also involved in the development of this project, since the methodology presented in this document is aligned with the objectives of this project, as it contributes to the runtime self-adaptation loop by evaluating different optimizing transformations and reconfiguring each application accordingly.
- **Plataforma HW/SW distribuida para el procesamiento inteligente de información sensorial heterogénea en aplicaciones de supervisión de grandes espacios naturales (PLATINO)**: PLATINO is a national project funded by the Spanish Ministry of Science, Innovation and Universities through the 2017 call of the National Research Program (TEC2017-86722-C4-4-R). This project is a collaboration among four Spanish universities, and its objective is to propose solutions to enhance the processing of heterogeneous sensor information with distributed hardware/software platforms. The scenario where these solutions are tested is the application of HSI technology to smart farming, where the use of multisensory networks is a breakthrough in key factors such as economy, food

quality and safety. In this context, the candidate has been deeply involved in applying the methodology developed in this PhD to increase the throughput of dataflow-based applications to process HSI, as described in Section 8.3.

9.3.3 Collaborations

The development of this PhD and the activities associated to it has encouraged the close collaboration with other institutions and research groups, either with the objective of fulfilling certain activities related to the research projects aforementioned, or to develop some of the solutions proposed in this document.

- **Collaboration with Instituto Universitario de Microelectrónica Aplicada (IUMA) research group from Universidad de Las Palmas de Gran Canaria (ULPGC):** In the context of HELICoiD, the candidate has maintained a close collaboration with this research group with the objective of devising efficient implementations of the algorithms conforming HELICoiD's processing chain so as to fulfill the performance requirements associated to this project. One of the main outcomes of this collaboration is the integration of the HSI processing chain into the demonstrator built as an outcome of the project. With respect to the work presented in this document, the contributions of this collaboration are twofold: first, it helped in defining the use-case chosen in this PhD –Section 7.2–; secondly, it led to the publication of 10 journals and 6 international conferences.
- **Collaboration with iCube laboratory from Université de Strasbourg:** The candidate carried out a **three-month research stay** with the Compilation pour les Architectures Multi-coeurS (CAMUS) team at iCube laboratory as part of the work presented in this PhD, which has led to a close collaboration of more than one year. This team is known for its expertise with polyhedral transformations, and is behind the development of Apollo, outlined in Section 4.1 and selected as one of the cornerstones of the solution proposed in this document. The aim of the stay was to study Apollo and design the extensions presented in Chapter 5 to make this tool amenable to dataflow-based applications. Some of the ground work to integrate this tool with a dataflow environment was also covered. This collaboration has led to 1 publication in an international conference.
- **Collaboration with Institut d'Électronique et de Télécommunications de Rennes (IETR) research group from Institut National des Sciences Appliquées (INSA):** The candidate has also collaborated with Video Analysis and Architecture Design for Embedded Resources (VAADER) team, from IETR, to extend PREESM and SPiDER to support polyhedral transformations through Apollo. This group has a vast background in dataflow MoCs and is responsible for the development of the aforementioned tools, which are the other cornerstones of the work developed in this PhD. This collaboration is aligned with CERBERO, as its objective is in line with this project, and INSA is one of the universities of the consortium. From this collaboration, a journal publication is envisioned.

9.3.4 Open-access products

Several open-source repositories and tutorials have been made available to grant the access to the products generated as a result of the work presented in this document.

- **Repositories:** The resources related to the different topics considered in this manuscript can be found in the following repositories:
 - **Apollo:** The multiversioning system presented in Chapter 5 can be found in <https://scm.gforge.inria.fr/anonscm/git/apollo>.
 - **PREESM and SPiDER:** Additionally, Apollo has been embedded in PREESM and SPiDER open-source frameworks (<https://github.com/preesm>).
 - **Benchmarks:** The dataflow-based versions of the benchmarks used in Chapter 8 to functionally validate the extensions to the Y-chart design methodology can be found in <https://gitlab.citsem.upm.es/raquel.lazcano/benchmarks>.
 - **HSI cancer detection application:** The dataflow version of the real use-case selected to validate the proposed extension –i.e., HELICoiD supervised classification stage with PCA, SVM and KNN algorithms– can be found in <https://gitlab.citsem.upm.es/raquel.lazcano/helicoidPreesm>. Likewise, the hyperspectral images used as inputs are available online¹.
- **Tutorials:** A tutorial has been created in order to learn how to use Apollo within PREESM and SPiDER. The tutorial is available on PREESM website (<https://preesm.github.io/tutos/apollo>).

9.3.5 Grants obtained

During the development of the work presented in this document, the candidate has obtained the following grants:

- **Contrato Predoctoral del Programa Propio RR01/2015.** This grant, awarded by Universidad Politécnica de Madrid (UPM), funded 4 years of the PhD.
- **Ayuda para la realización de una Estancia de Investigación Internacional del Programa Propio** grant given by UPM. This grant provided financial support for the 3-month stay of the candidate at Université de Strasbourg.

¹<https://hsibraindatabase.iuma.ulpgc.es/>

9.3.6 Awards received

During the development of the work presented in this document, the candidate has received the following awards:

- **DASIP 2016 Best Demo Award.** [Salvador'16] R. Salvador, H. Fabelo, R. Lazcano, S. Ortega, D. Madroñal, G. M. Callicó, E. Juárez, C. Sanz, "HELICoiD tool demonstrator for real-time brain cancer detection", in *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 237–238, IEEE, 2016.
- **Best Student Paper Award.** [Lazcano'17a] R. Lazcano, D. Madroñal, H. Fabelo, S. Ortega, R. Salvador, G. Callicó, E. Juárez, C. Sanz, "Parallel exploitation of a spatial-spectral classification approach for hyperspectral images on RVC-CAL", in *High-Performance Computing in Geoscience and Remote Sensing VII*, vol. 10430, p. 104300A, International Society for Optics and Photonics, 2017.
- **Primer Premio UPM Innovatech 2T Challenge 2018.** This award was obtained with a technology based on the outcomes of HELICoiD project.

9.4 Future research lines

This document has proposed, described and evaluated a methodology to further optimize applications by combining the parallelization capabilities of Y-chart dataflow-based design with the optimization potential offered by polyhedral-based frameworks. To assess this methodology, a prototype has been built by (i) extending a polyhedral tool from the state of the art, Apollo, to be compliant with dataflow-based applications; and (ii) embedding this extended version within a Y-chart dataflow-based design framework, such as PREESM. This prototype has been validated through a custom, heterogeneous benchmark suite, and its applicability to a real-life problem has been demonstrated by designing a real use-case following this methodology.

However, as already hinted throughout the document, there are still several aspects of this work subject to improvement. These improvements can be grouped into three different categories, attending to the field of application: first, Subsection 9.4.1 introduces the future research lines related to extensions that can be added within a polyhedral-based framework, such as Apollo, to further exploit the features intrinsic to dataflow; afterwards, Subsection 9.4.2 presents the enhancements envisioned in the methodology to promote its advantages; finally, Subsection 9.4.3 closes this document by outlining the improvements devised for the HSI cancer detection application.

9.4.1 Runtime extensions of the polyhedral model

As described in this document, one of the bases upon which the proposed methodology has been built is the extension of Apollo's runtime to adapt the transformation applied to the execution context. Although its advantages have already been proven, there are several improvements that are easily foreseen.

Up until now, the extensions addressed in this document have only involved Apollo's dynamic phase, i.e., its runtime manager. Yet, as advanced in Subsection 8.2.1, Apollo's static component can be also modified to help in increasing the efficiency of the methodology. Specifically, the results obtained showed that, depending on several factors, the same transformation could yield very efficient results or, on the contrary, rather inefficient ones, which resulted in adding a non-negligible overhead in the training phase. One of the reasons of this overhead is that, once the transformation has been generated and launched, its execution cannot be stopped before its completion; hence, if the transformation generated is very inefficient, this execution will add a useless overhead, since it is obvious that this transformation will not be chosen as the best performing one. Consequently, this future research line proposes adding another timeout, this time for the execution of the transformation under evaluation: if, when compared to the previously stored version –which can be either a transformation or the original code–, the execution time of this transformation is larger, this execution can be preempted and the original code –or the stored transformation– launched for the remaining set of iterations, thus automatically discarding the inefficient one.

Continuing with the efficiency of the tested transformations, the rationale followed to build the transformation set for evaluating the tool has been trying to have a set as heterogeneous as possible, conjugating different optimization opportunities –such as different tile sizes or different unrolling factors–, but there are many more possibilities. With the flags offered by Pluto to control the transformation, the range of possibilities is rather large. As advanced in Section 8.1, a very interesting research line is to carry out a so-called Transformation Space Exploration (TSE), which consists of conducting a thorough and detailed characterization of the impact each configuring option has in the overall execution time, trying to link their performance to features such as architecture characteristics or volume of data involved in the processing, with the objective of directly devising the most efficient transformations, without having to follow a trial-and-error strategy. This study leads, in turn, to another research line: as already stated, finding the best performing transformation –or set of transformations– is not an easy task, and involves studying many possibilities; in this context, machine learning techniques may come in handy, since they can be of great help in extracting a pattern to choose these transformations. Yet, machine learning algorithms usually involve very time-consuming and complex tasks, so devising an efficient way to embed this kind of algorithms within a runtime manager is another research line by itself.

Following the idea of reducing the overhead added by the training phase of the multiversioning system, another research line is proposed in this context, which basically consists of offloading the results of the training phase to a file in the hard disk.

By doing so, the conclusions extracted from the training phase would be maintained even if the execution is aborted and relaunched, which in turn would mean turning this training into an offline process that can be executed only once, thus directly providing the results of the operational phase, free from the mentioned overhead.

To conclude this section, the last research line that can emerge from this PhD in this context is the extension of Apollo to be supported in embedded architectures, such as the one mentioned in the use-case –MPPA-256-N–. As already described, one of the main advantages of a tool like PREESM is that it offers the possibility of automatically generating code for many different target architectures, not only to x86 ones. Hence, to improve Apollo as a standalone tool, but also to extend the applicability of the methodology proposed in this document, another research line proposed is to ensure the support of Apollo, at least, in the same architectures PREESM already supports.

9.4.2 Extensions to the proposed methodology

The experiments presented in this document have proven the advantages a methodology such as the one proposed may offer to time-consuming applications, but there are several aspects that can be considered to improve it.

As already mentioned, the creation of an API opened a wide range of opportunities, since it establishes the bases for communicating both worlds. This communication involves passing to one domain or the other all the information already described, but it also enables many more possibilities. As already established in Section 6.1, `apolloStopTxSelection()` function was added to the API to illustrate the additional possibilities it can offer. In the case of this function, it was devised to grant PREESM –or SPiDER– even more control over Apollo by deciding when to stop the training phase. This function, coupled with `apolloGetEstimatedTime()`, implemented a mechanism by which, after the execution of each iteration of the application graph, Apollo sent back an estimation of the execution time associated to that transformation for the whole kernel; if that execution time was considered enough, then the training phase was aborted, and the operational phase directly launched, hence reducing the overhead of the training phase. This is just an example of the potential offered when adding new functions to the API, which is the proposed research line.

The last emerging research line in this context is related to combining the parallelization potentials of polyhedral and dataflow domains. As already said, the envisioned interferences between them have been solved by granting this potential to one or the other, and deactivating Apollo’s parallelization potential when coupled to PREESM, turning it into just an optimizer. Yet, one of the most interesting research lines emerging from this work is precisely studying how to actively combine both potentials, so as to make the methodology more flexible by allowing the support of heterogeneous parallelization schemes, envisioning scenarios such as dividing the PEs between both tools; in this situation, PREESM could use half of them to exploit the structural parallelization, while Apollo uses the other half to perform aggressive transformations, exploiting thus inter- and intra-actor parallelization. To allow such

a behavior, a global manager to handle this kind of heterogeneous and hierarchical scheduling must be devised, since, up until now, each tool has its own scheduler.

9.4.3 HSI cancer detection application

Finally, the last research lines emerging from this PhD that are going to be presented are those related to the use-case. Although this application was born as a result of an international project already finished, the research associated to it has continued, and hence new possibilities are susceptible of being explored.

As stated while describing the use-case, the supervised classification stage was selected to be offloaded to a hardware accelerator within HELICoiD project due to its large computational load, which is also the reason why this is the part that has been considered as the use-case of this PhD. However, that does not mean the rest of the parts conforming the algorithm cannot be targeted by a methodology such as the one proposed in this document; as a matter of fact, the supervised classification stage is now faster than the rest of the algorithm, as stated in [Fabelo'18b]. For that reason, one of the future research lines proposed in this context is to apply the methodology not only to the supervised classification stage, but also to the unsupervised one, as well as to the preprocessing of the image and the majority voting step.

Related also to this topic, the results obtained showed that there were some parts of the algorithm that could not be targeted by Apollo, mainly because the behavior they represented did not follow a linear model, and thus a prediction model could not be inferred. In this line, it is also worth reviewing why those cases could not build a prediction model, in case they can be modified to successfully build it or, as a last resort, to explore the possibility of substituting the algorithms by similar ones. For example, this can be a good solution for KNN: for each pixel, it applies some processing to a number of neighbors that need to be found before; hence, this processing depends on the memory positions of those neighbors, which cannot be predicted. In this regard, some preliminary works are starting to consider the possibility of substituting KNN by a spatial filter in which the neighbors are predefined; with this new filter, Apollo is now supposed to be able to build a prediction model, and thus improve the execution time of this part of the algorithm, which was one of the main bottlenecks.

Bibliography

- [Acar'03] U. A. Acar, G. E. Blelloch, R. Harper, "Selective Memoization", in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pp. 14–25, ACM, New York, NY, USA, 2003, ISBN 1-58113-628-5, doi:10.1145/604131.604133, <http://doi.acm.org/10.1145/604131.604133>.⁷⁹
- [Ackerman'79] W. B. Ackerman, "Data flow languages", in *1979 International Workshop on Managing Requirements Knowledge (MARK)*, pp. 1087–1095, IEEE, 1979.¹⁹
- [Akbari'10] H. Akbari, Y. Kosugi, K. Kojima, N. Tanaka, "Detection and analysis of the intestinal ischemia using visible and invisible hyperspectral imaging", in *IEEE Transactions on Biomedical Engineering*, vol. 57, no. 8, pp. 2011–2017, 2010.³¹
- [Akbari'11] H. Akbari, K. Uto, Y. Kosugi, K. Kojima, N. Tanaka, "Cancer detection using infrared hyperspectral imaging", in *Cancer science*, vol. 102, no. 4, pp. 852–857, 2011.³³
- [Aragón'19] J. S. Aragón, S. S. Ramírez, R. L. López, D. M. Quintín, R. S. Perea, E. J. Martínez, C. S. Álvaro, "Characterizing Hyperspectral Data Layouts: Performance and Energy Efficiency in Embedded GPUs for PCA-based Dimensionality Reduction", in *2019 XXXIV Conference on Design of Circuits and Integrated Systems (DCIS)*, pp. 1–6, IEEE, 2019.²¹⁵
- [Bacon'94] D. F. Bacon, S. L. Graham, O. J. Sharp, "Compiler transformations for high-performance computing", in *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.^{49, 50}
- [Baghdadi'15] R. Baghdadi, U. Beaunon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, et al., "Pencil: A platform-neutral compute intermediate language for accelerator programming", in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 138–149, IEEE, 2015.²⁴
- [Baghdadi'19] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code", in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 193–205, IEEE Press, 2019.²⁴
- [Ball'65] G. H. Ball, D. J. Hall, "ISODATA, a novel method of data analysis and pattern classification", Tech. rep., Stanford research inst Menlo Park CA, 1965.³⁷

BIBLIOGRAPHY

- [Baltussen'19] E. J. Baltussen, E. N. Kok, S. G. B. de Koning, J. Sanders, A. G. Aalbers, N. F. Kok, G. L. Beets, C. C. Flohil, S. C. Bruin, K. F. Kuhlmann, et al., "Hyperspectral imaging for tissue classification, a way toward smart laparoscopic colorectal surgery", in *Journal of biomedical optics*, vol. 24, no. 1, p. 016 002, 2019. ³³
- [Bastoul'04] C. Bastoul, "Code generation in the polyhedral model is easier than you think", in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 7–16, IEEE Computer Society, 2004. ²⁴
- [Bastoul'11] C. Bastoul, "Openscop: A specification and a library for data exchange in polyhedral compilation tools", in *Paris-Sud University, France, Tech. Rep.*, vol. 9, 2011. ²²
- [Belikov'13] E. Belikov, P. Deligiannis, P. Tooto, M. Aljabri, H.-W. Loidl, "A survey of high-level parallel programming models", in *Heriot-Watt University, Edinburgh, UK*, vol. 1, no. 2, pp. 2–2, 2013. ¹⁵
- [Benediktsson'05] J. A. Benediktsson, J. A. Palmason, J. R. Sveinsson, "Classification of hyperspectral data from urban areas based on extended morphological profiles", in *IEEE Transactions on Geoscience and Remote Sensing*, vol. 43, no. 3, pp. 480–491, 2005. ³⁶
- [Bhattacharyya'06] S. S. Bhattacharyya, W. S. Levine, "Optimization of signal processing software for control system implementation", in *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pp. 1562–1567, IEEE, 2006. ¹⁰
- [Bjorgan'15] A. Bjorgan, M. Denstedt, M. Milanič, L. A. Paluchowski, L. L. Randeberg, "Vessel contrast enhancement in hyperspectral images", in *Optical Biopsy XIII: Toward Real-Time Spectroscopic Imaging and Diagnosis*, vol. 9318, p. 93180G, International Society for Optics and Photonics, 2015. ³¹
- [Bondhugula'08] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer", in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, pp. 101–113, ACM, New York, NY, USA, 2008, ISBN 978-1-59593-860-2, doi:10.1145/1375581.1375595, <http://doi.acm.org/10.1145/1375581.1375595>. ^{13, 21, 22, 48}
- [Bondhugula'10] U. Bondhugula, S. Dash, O. Gunluk, L. Renganarayanan, "A model for fusion and code motion in an automatic parallelizing compiler", in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 343–352, IEEE, 2010. ²³
- [Butenhof'97] D. R. Butenhof, *Programming with POSIX threads*, Addison-Wesley Professional, 1997. ¹⁶

-
- [Caamaño'16] J. M. M. Caamaño, *Fast and flexible compilation techniques for effective speculative polyhedral parallelization*, Ph.D. thesis, 2016. 45, 47, 52, 53, 54, 55, 56, 111, 113, 126, 129
- [Caamaño'17] J. M. M. Caamaño, M. Selva, P. Clauss, A. Baloian, W. Wolff, "Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones", in *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, p. e4192, 2017, doi: 10.1002/cpe.4192, <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4192>. 26, 53, 68
- [Calin'14] M. A. Calin, S. V. Parasca, D. Savastru, D. Manea, "Hyperspectral imaging in the medical field: Present and future", in *Applied Spectroscopy Reviews*, vol. 49, no. 6, pp. 435–447, 2014. 31
- [Campbell'06] S. L. Campbell, J.-P. Chancelier, R. Nikoukhah, *Modeling and Simulation in SCILAB*, Springer, 2006. 24
- [Camps-Valls'05] G. Camps-Valls, L. Bruzzone, "Kernel-based methods for hyperspectral image classification", in *IEEE Transactions on Geoscience and Remote Sensing*, vol. 43, no. 6, pp. 1351–1362, 2005. 36, 119
- [Carvalho Rocha'11] W. F. de Carvalho Rocha, G. P. Sabin, P. H. Março, R. J. Poppi, "Quantitative analysis of piroxicam polymorphs pharmaceutical mixtures by hyperspectral imaging and chemometrics", in *Chemometrics and Intelligent Laboratory Systems*, vol. 106, no. 2, pp. 198–204, 2011. 31
- [Ceng'08] J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, H. Kunieda, "MAPS: an integrated framework for MPSoC application parallelization", in *Proceedings of the 45th annual Design Automation Conference*, pp. 754–759, ACM, 2008. 19
- [Chacko'03] A. Chacko, N. Kumar, G. Chacko, R. Athyal, V. Rajshekhar, "Intraoperative ultrasound in determining the extent of resection of parenchymal brain tumours—a comparative study with computed tomography and histopathology", in *Acta neurochirurgica*, vol. 145, no. 9, pp. 743–748, 2003. 35
- [Chan'08] J. C.-W. Chan, D. Paelinckx, "Evaluation of Random Forest and Adaboost tree-based ensemble classification and spectral band selection for ecotope mapping using airborne hyperspectral imagery", in *Remote Sensing of Environment*, vol. 112, no. 6, pp. 2999–3011, 2008. 36
- [Chandra'01] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, *Parallel programming in OpenMP*, Morgan kaufmann, 2001. 17
- [Chang'03] C.-I. Chang, *Hyperspectral imaging: techniques for spectral detection and classification*, vol. 1, Springer Science & Business Media, 2003. 36
- [Che'09] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing", in *2009 IEEE*

BIBLIOGRAPHY

- international symposium on workload characterization (IISWC)*, pp. 44–54, Ieee, 2009. ^{29, 113}
- [Chen'08] C. Chen, J. Chame, M. Hall, "CHiLL: A framework for composing high-level loop transformations", Tech. rep., Citeseer, 2008. ²³
- [Chowdhury'18] T. Chowdhury, F. A. Zeiler, G. P. Singh, A. Hailu, H. Loewen, B. Schaller, R. B. Cappellani, M. West, "The role of intraoperative MRI in awake neurosurgical procedures: a systematic review", in *Frontiers in oncology*, vol. 8, p. 434, 2018. ³⁵
- [Clauss'96] P. Clauss, "Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs", in *Proceedings of the 10th International Conference on Supercomputing*, ICS '96, pp. 278–285, New York, NY, USA, 1996. ⁷⁵
- [Consel'96] C. Consel, F. Noël, "A General Approach for Run-time Specialization and Its Application to C", in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pp. 145–156, ACM, New York, NY, USA, 1996, ISBN 0-89791-769-3, doi: 10.1145/237721.237767, <http://doi.acm.org/10.1145/237721.237767>. ⁸⁴
- [Dagum'98] L. Dagum, R. Menon, "OpenMP: An industry-standard API for shared-memory programming", in *Computing in Science & Engineering*, , no. 1, pp. 46–55, 1998. ¹⁶
- [Dale'13] L. M. Dale, A. Thewis, C. Boudry, I. Rotar, P. Dardenne, V. Baeten, J. A. F. Pierna, "Hyperspectral imaging applications in agriculture and agro-food product quality and safety control: a review", in *Applied Spectroscopy Reviews*, vol. 48, no. 2, pp. 142–159, 2013. ³¹
- [Desnos'13] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, S. Aridhi, "Pimm: Parameterized and interfaced dataflow meta-model for mpsoes runtime reconfiguration", in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 41–48, IEEE, 2013. ¹²
- [Dey'10] V. Dey, Y. Zhang, M. Zhong, *A review on image segmentation techniques with remote sensing perspective*, vol. 38, na Vienna, Austria, 2010. ³⁶
- [Dinechin'13] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, et al., "A clustered manycore processor architecture for embedded and accelerated applications", in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2013. ¹⁷⁶
- [Doerfert'13] J. Doerfert, C. Hammacher, K. Streit, S. Hack, "Spolly: speculative optimizations in the polyhedral model", in *IMPACT 2013*, vol. 55, 2013. ²⁵

- [Dolbeau'07] R. Dolbeau, S. Bihan, F. Bodin, "HMPP: A hybrid multi-core parallel programming environment", in *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, vol. 28, 2007. ¹⁷
- [Domingo'17] R. Domingo, R. Salvador, H. Fabelo, D. Madroñal, S. Ortega, R. Lazcano, E. Juárez, G. Callicó, C. Sanz, "High-level design using Intel FPGA OpenCL: A hyperspectral imaging spatial-spectral classifier", in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, IEEE, 2017. ²¹⁶
- [Duncan'90] R. Duncan, "A survey of parallel computer architectures", in *Computer*, vol. 23, no. 2, pp. 5–16, 1990. ¹⁵
- [Edelman'12] G. Edelman, E. Gaston, T. Van Leeuwen, P. Cullen, M. Aalders, "Hyperspectral imaging for non-contact analysis of forensic traces", in *Forensic science international*, vol. 223, no. 1-3, pp. 28–39, 2012. ³¹
- [Edwards'97] S. Edwards, L. Lavagno, E. A. Lee, A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis", in *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, 1997. ¹
- [El Rahman'15] S. A. El Rahman, "Hyperspectral imaging classification using ISODATA algorithm: Big data challenge", in *2015 Fifth International Conference on e-Learning (econf)*, pp. 247–250, IEEE, 2015. ³⁷
- [Fabelo'16a] H. Fabelo, S. Ortega, R. Guerra, G. M. Callicó, A. Szolna, J. F. Piñeiro, M. Tejedor, S. López, R. Sarmiento, "A Novel Use of Hyperspectral Images for Human Brain Cancer Detection using in-Vivo Samples.", in *BIOSIGNALS*, pp. 311–320, 2016. ^{116, 190}
- [Fabelo'16b] H. Fabelo, S. Ortega, S. Kabwama, G. M. Callico, D. Bulters, A. Szolna, J. F. Pineiro, R. Sarmiento, "HELICoiD project: A new use of hyperspectral imaging for brain cancer detection in real-time during neurosurgical operations", in *Hyperspectral Imaging Sensors: Innovative Applications and Sensor Standards 2016*, vol. 9860, p. 986002, International Society for Optics and Photonics, 2016. ³⁵
- [Fabelo'18a] H. Fabelo, S. Ortega, R. Lazcano, D. Madroñal, G. M Callicó, E. Juárez, R. Salvador, D. Bulters, H. Bulstrode, A. Szolna, et al., "An intraoperative visualization system using hyperspectral imaging to aid in brain tumor delineation", in *Sensors*, vol. 18, no. 2, p. 430, 2018. ^{35, 114, 214}
- [Fabelo'18b] H. Fabelo, S. Ortega, D. Ravi, B. R. Kiran, C. Sosa, D. Bulters, G. M. Callicó, H. Bulstrode, A. Szolna, J. F. Piñeiro, et al., "Spatio-spectral classification of hyperspectral images for brain cancer detection during surgical operations", in *PLoS One*, vol. 13, no. 3, 2018. ^{35, 116, 118, 120, 176, 187, 214, 224}
- [Fabelo'19a] H. Fabelo, M. Halicek, S. Ortega, M. Shahedi, A. Szolna, J. F. Piñeiro, C. Sosa, A. J. O'Shanahan, S. Bisshopp, C. Espino, et al., "Deep learning-based framework for in vivo identification of glioblastoma tumor using

BIBLIOGRAPHY

- hyperspectral images of human brain”, in *Sensors*, vol. 19, no. 4, p. 920, 2019. ³⁵
- [Fabelo’19b] H. Fabelo, S. Ortega, A. Szolna, D. Bulters, J. E. Piñeiro, S. Kabwama, A. JO’Shanahan, H. Bulstrode, S. Bisshopp, B. R. Kiran, et al., “In-vivo hyperspectral human brain image database for brain cancer detection”, in *IEEE Access*, vol. 7, pp. 39 098–39 116, 2019. ^{35, 115, 177, 190, 213}
- [Fauvel’12] M. Fauvel, Y. Tarabalka, J. A. Benediktsson, J. Chanussot, J. C. Tilton, “Advances in spectral-spatial classification of hyperspectral images”, in *Proceedings of the IEEE*, vol. 101, no. 3, pp. 652–675, 2012. ³⁶
- [Feautrier’88] P. Feautrier, “Parametric integer programming”, in *RAIRO-Operations Research*, vol. 22, no. 3, pp. 243–268, 1988. ⁹⁶
- [Feautrier’11] P. Feautrier, C. Lengauer, “Polyhedron Model”, in D. Padua, editor, *Encyclopedia of Parallel Computing*, pp. 1581–1592, Springer US, 2011, ISBN 978-0-387-09765-7, doi:10.1007/978-0-387-09766-4. ^{13, 15, 21, 48}
- [Fei’17] B. Fei, G. Lu, X. Wang, H. Zhang, J. V. Little, M. R. Patel, C. C. Griffith, M. W. El-Diery, A. Y. Chen, “Label-free reflectance hyperspectral imaging for tumor margin assessment: a pilot study on surgical specimens of cancer patients”, in *Journal of biomedical optics*, vol. 22, no. 8, p. 086 009, 2017. ³⁴
- [Fleming’86] P. J. Fleming, J. J. Wallace, “How not to lie with statistics: the correct way to summarize benchmark results”, in *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, 1986. ²⁷
- [Floc’h’13] A. Floc’h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L’Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, O. Sentieys, “GeCoS: A framework for prototyping custom hardware design flows”, in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pp. 100–105, Sept 2013. ²⁴
- [Florimbi’18] G. Florimbi, H. Fabelo, E. Torti, R. Lazcano, D. Madroñal, S. Ortega, R. Salvador, F. Leporati, G. Danese, A. Báez-Quevedo, et al., “Accelerating the K-nearest neighbors filtering algorithm to optimize the real-time classification of human brain tumor in hyperspectral images”, in *Sensors*, vol. 18, no. 7, p. 2314, 2018. ^{115, 120, 176, 187, 213}
- [Fox’14] G. C. Fox, R. D. Williams, G. C. Messina, *Parallel computing works!*, Elsevier, 2014. ²⁰
- [Gajinov’15] V. Gajinov, S. Stipić, I. Erić, O. S. Unsal, E. Ayguadé, A. Cristal, “DaSH: A benchmark suite for hybrid dataflow and shared memory programming models”, in *Parallel Computing*, vol. 45, pp. 18–48, 2015. ²⁸
- [Georgakarakos’17] G. Georgakarakos, S. Kanur, J. Lilius, K. Desnos, “Task-based execution of synchronous dataflow graphs for scalable multicore computing”, in *2017*

- IEEE International Workshop on Signal Processing Systems (SiPS)*, pp. 1–6, IEEE, 2017. ^{60, 110}
- [Gerard'17] I. J. Gerard, M. Kersten-Oertel, K. Petrecca, D. Sirhan, J. A. Hall, D. L. Collins, "Brain shift in neuronavigation of brain tumors: A review", in *Medical image analysis*, vol. 35, pp. 403–420, 2017. ³⁵
- [Gilles'74] K. Gilles, "The semantics of a simple language for parallel programming", in *Information processing*, vol. 74, pp. 471–475, 1974. ¹³
- [Giorgi'16] R. Giorgi, "Exploring dataflow-based thread level parallelism in cyber-physical systems", in *Proceedings of the ACM International Conference on Computing Frontiers*, pp. 295–300, 2016. ¹¹⁰
- [Giua'07] A. Giua, C. Seatzu, "A systems theory view of Petri nets", in *Advances in control theory and applications*, pp. 99–127, Springer, 2007. ¹⁴
- [Govender'07] M. Govender, K. Chetty, H. Bulcock, "A review of hyperspectral remote sensing and its application in vegetation and water resource studies", in *Water Sa*, vol. 33, no. 2, 2007. ³¹
- [Grandpierre'99] T. Grandpierre, C. Lavarenne, Y. Sorel, "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors", in *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99)*(IEEE Cat. No. 99TH8450), pp. 74–78, IEEE, 1999. ¹⁹
- [Grant'99] B. Grant, M. Philipose, M. Mock, C. Chambers, S. J. Eggers, "An Evaluation of Staged Run-time Optimizations in DyC", in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pp. 293–304, ACM, New York, NY, USA, 1999, ISBN 1-58113-094-5, doi:10.1145/301618.301683, <http://doi.acm.org/10.1145/301618.301683>. ⁸⁴
- [Grosser'12] T. Grosser, A. Groesslinger, C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation", in *Parallel Processing Letters*, vol. 22, no. 04, p. 1250 010, 2012. ²²
- [Gu'19] R. Gu, M. Becchi, "A comparative study of parallel programming frameworks for distributed GPU applications", in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pp. 268–273, 2019. ¹⁵
- [Halicek'17] M. Halicek, G. Lu, J. V. Little, X. Wang, M. Patel, C. C. Griffith, M. W. El-Deiry, A. Y. Chen, B. Fei, "Deep convolutional neural networks for classifying head and neck cancer using hyperspectral imaging", in *Journal of biomedical optics*, vol. 22, no. 6, p. 060 503, 2017. ³⁴
- [Halicek'19] M. Halicek, H. Fabelo, S. Ortega, G. M. Callico, B. Fei, "In-vivo and ex-vivo tissue analysis through hyperspectral imaging techniques: Revealing the invisible features of cancer", in *Cancers*, vol. 11, no. 6, p. 756, 2019. ^{30, 31, 32, 34, 36, 37}

BIBLIOGRAPHY

- [Hall'97] M. Hall, J. P. McNamee, "Improving software performance with automatic memoization", in *Johns Hopkins APL Technical Digest*, vol. 18, no. 2, p. 255, 1997. ⁷⁹
- [Han'16] Z. Han, A. Zhang, X. Wang, Z. Sun, M. D. Wang, T. Xie, "In vivo use of hyperspectral imaging to develop a noncontact endoscopic diagnosis support system for malignant colorectal tumors", in *Journal of biomedical optics*, vol. 21, no. 1, p. 016 001, 2016. ³³
- [Heinecke'13] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, P. Dubey, "Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor", in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 126–137, IEEE, 2013. ¹¹⁰
- [Hemeida'20] A. Hemeida, S. Hassan, S. Alkhalaf, M. Mahmoud, M. Saber, A. M. B. Eldin, T. Senjyu, A. H. Alayed, "Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor", in *Ain Shams Engineering Journal*, 2020. ¹¹⁰
- [Hernández-Vila'16] C. Hernández-Vila, "Current prognosis and quality of life following surgical treatment for head and neck squamous cell carcinoma", in *Plast. Aesthetic Res*, vol. 3, p. 203, 2016. ³⁴
- [Heulot'14] J. Heulot, M. Pelcat, K. Desnos, J.-F. Nezan, S. Aridhi, "Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps", in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pp. 167–171, IEEE, 2014. ^{20, 63}
- [Hinton'09] G. E. Hinton, "Deep belief networks", in *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009. ³⁷
- [Huang'16] K. Huang, S. Li, X. Kang, L. Fang, "Spectral-spatial hyperspectral image classification based on KNN", in *Sensing and Imaging*, vol. 17, no. 1, p. 1, 2016. ¹²⁰
- [Hussain'13] H. Hussain, S. U. R. Malik, A. Hameed, S. U. Khan, G. Bickler, N. Min-Allah, M. B. Qureshi, L. Zhang, W. Yongji, N. Ghani, et al., "A survey on resource allocation in high performance distributed computing systems", in *Parallel Computing*, vol. 39, no. 11, pp. 709–736, 2013. ¹⁵
- [Hwang'13] E. S. Hwang, D. Y. Lichtensztajn, S. L. Gomez, B. Fowble, C. A. Clarke, "Survival after lumpectomy and mastectomy for early stage invasive breast cancer: the effect of age and hormone receptor status", in *Cancer*, vol. 119, no. 7, pp. 1402–1411, 2013. ³³
- [Jacobs'08] L. Jacobs, "Positive margins: the challenge continues for breast surgeons", in *Annals of surgical oncology*, vol. 15, no. 5, pp. 1271–1272, 2008. ³³
- [Jain'10] A. K. Jain, "Data clustering: 50 years beyond K-means", in *Pattern recognition letters*, vol. 31, no. 8, pp. 651–666, 2010. ³⁷

- [Jayanthi'11] J. Jayanthi, G. Nisha, S. Manju, E. Philip, P. Jeemon, K. Baiju, V. Beena, N. Subhash, "Diffuse reflectance spectroscopy: diagnostic accuracy of a non-invasive screening technique for early detection of malignant changes in the oral cavity", in *BMJ open*, vol. 1, no. 1, p. e000 071, 2011.³⁴
- [Kamruzzaman'16] M. Kamruzzaman, D.-W. Sun, "Introduction to hyperspectral imaging technology", in *Computer vision technology for food quality evaluation*, pp. 111–139, Elsevier, 2016.³⁰
- [Kasim'08] H. Kasim, V. March, R. Zhang, S. See, "Survey on parallel programming model", in *IFIP International Conference on Network and Parallel Computing*, pp. 266–275, Springer, 2008.¹⁵
- [Kessler'07] C. Kessler, J. Keller, "Models for parallel computing: Review and perspectives", in *Mitteilungen-Gesellschaft für Informatik eV, Parallel-Algorithmen und Rechnerstrukturen*, vol. 24, pp. 13–29, 2007.¹⁵
- [Kienhuis'00] B. Kienhuis, E. Rijpkema, E. Deprettere, "Compaan: Deriving process networks from matlab for embedded signal processing architectures", in *Proceedings of the eighth international workshop on Hardware/software codesign*, pp. 13–17, 2000.²⁰
- [Kienhuis'01] B. Kienhuis, E. F. Deprettere, P. Van der Wolf, K. Vissers, "A methodology to design programmable embedded systems", in *International Workshop on Embedded Computer Systems*, pp. 18–37, Springer, 2001.^{1, 19}
- [Kim'13] B. Kim, N. Kehtarnavaz, P. LeBoulluec, H. Liu, Y. Peng, D. Euhus, "Automation of ROI extraction in hyperspectral breast images", in *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 3658–3661, IEEE, 2013.³³
- [Kiyotoki'13] S. Kiyotoki, J. Nishikawa, T. Okamoto, K. Hamabe, M. Saito, A. Goto, Y. Fujita, Y. Hamamoto, Y. Takeuchi, S. Satori, et al., "New method for detection of gastric cancer by hyperspectral imaging: a pilot study", in *Journal of biomedical optics*, vol. 18, no. 2, p. 026 010, 2013.³¹
- [Korbar'17] B. Korbar, A. M. Olofson, A. P. Miraflor, C. M. Nicka, M. A. Suriawinata, L. Torresani, A. A. Suriawinata, S. Hassanpour, "Deep learning for classification of colorectal polyps on whole-slide images", in *Journal of pathology informatics*, vol. 8, 2017.³⁷
- [Kruse'93] F. A. Kruse, A. Lefkoff, J. Boardman, K. Heidebrecht, A. Shapiro, P. Barloon, A. Goetz, "The spectral image processing system (SIPS)-interactive visualization and analysis of imaging spectrometer data", in *AIP Conference Proceedings*, vol. 283, pp. 192–201, American Institute of Physics, 1993.¹¹⁶
- [Lazcano'16a] R. Lazcano, D. Madroñal, K. Desnos, M. Pelcat, R. Guerra, S. López, E. Juárez, C. Sanz, "Parallelism Exploitation of a Dimensionality Reduction Algorithm Applied to Hyperspectral Images", in *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2016.²¹⁷

BIBLIOGRAPHY

- [Lazcano'16b] R. Lazcano, I. Sidrach-Cardona, D. Madroñal, K. Desnos, M. Pelcat, E. Juárez, C. Sanz, "Parallelism exploitation of a PCA algorithm for hyperspectral images using RVC-CAL", in *High-Performance Computing in Geoscience and Remote Sensing VI*, vol. 10007, p. 100070H, International Society for Optics and Photonics, 2016. ²¹⁷
- [Lazcano'17a] R. Lazcano, D. Madroñal, H. Fabelo, S. Ortega, R. Salvador, G. Callicó, E. Juárez, C. Sanz, "Parallel exploitation of a spatial-spectral classification approach for hyperspectral images on RVC-CAL", in *High-Performance Computing in Geoscience and Remote Sensing VII*, vol. 10430, p. 104300A, International Society for Optics and Photonics, 2017. ^{216, 221}
- [Lazcano'17b] R. Lazcano, D. Madroñal, H. Fabelo, S. Ortega, R. Salvador, G. M. Callicó, E. Juárez, C. Sanz, "Parallel implementation of an iterative PCA algorithm for hyperspectral images on a manycore platform", in *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1–6, IEEE, 2017. ²¹⁶
- [Lazcano'17c] R. Lazcano, D. Madroñal, R. Salvador, K. Desnos, M. Pelcat, R. Guerra, H. Fabelo, S. Ortega, S. López, G. M. Callicó, et al., "Porting a PCA-based hyperspectral image dimensionality reduction algorithm for brain cancer detection on a manycore architecture", in *Journal of Systems Architecture*, vol. 77, pp. 101–111, 2017. ^{119, 120, 176, 185, 214}
- [Lazcano'19a] R. Lazcano, D. Madroñal, H. Fabelo, S. Ortega, R. Salvador, G. Callico, E. Juarez, C. Sanz, "Adaptation of an iterative PCA to a manycore architecture for hyperspectral image processing", in *Journal of Signal Processing Systems*, vol. 91, no. 7, pp. 759–771, 2019. ^{176, 213}
- [Lazcano'19b] R. Lazcano, D. Madroñal, G. Florimbi, J. Sancho, S. Sanchez, R. Leon, H. Fabelo, S. Ortega, E. Torti, R. Salvador, et al., "Parallel Implementations Assessment of a Spatial-Spectral Classifier for Hyperspectral Clinical Applications", in *IEEE Access*, vol. 7, pp. 152 316–152 333, 2019. ^{xxv, xxviii, 35, 118, 176, 181, 182, 190, 198, 199, 201, 212}
- [Lazcano'20] R. Lazcano, D. Madroñal, E. Juarez, P. Clauss, "Runtime multi-versioning and specialization inside a memoized speculative loop optimizer", in *Proceedings of the 29th International Conference on Compiler Construction*, pp. 96–107, 2020. ^{75, 79, 215}
- [Le Gall'14] F. Le Gall, "Powers of tensors and fast matrix multiplication", in *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pp. 296–303, 2014. ¹¹⁰
- [LeCun'98] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-based learning applied to document recognition", in *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. ³⁷
- [LeCun'15] Y. LeCun, Y. Bengio, G. Hinton, "Deep learning", in *nature*, vol. 521, no. 7553, pp. 436–444, 2015. ³⁷

-
- [Lee'87] E. A. Lee, D. G. Messerschmitt, "Synchronous data flow", in *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. ^{1, 12}
- [Lee'95] E. A. Lee, T. M. Parks, "Dataflow process networks", in *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995. ¹³
- [Li'14a] M. Li, S. Zang, B. Zhang, S. Li, C. Wu, "A review of remote sensing image classification techniques: The role of spatio-contextual information", in *European Journal of Remote Sensing*, vol. 47, no. 1, pp. 389–411, 2014. ^{36, 37}
- [Li'14b] T. Li, J. Zhang, Y. Zhang, "Classification of hyperspectral image based on deep belief networks", in *2014 IEEE international conference on image processing (ICIP)*, pp. 5132–5136, IEEE, 2014. ³⁷
- [Liang'12] H. Liang, "Advances in multispectral and hyperspectral imaging for archaeology and art conservation", in *Applied Physics A*, vol. 106, no. 2, pp. 309–323, 2012. ³¹
- [Linck'06] P. Linck, E. Priedane, D. Hughes, R. Edwards, "Improving Outcomes for People with Brain and Other CNS Tumours: Analysis of the Potential Economic Impact of the Guidance.", in , 2006. ³⁴
- [Liu'06] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, J. Torrellas, "POSH: a TLS compiler that exploits program structure", in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 158–167, ACM, 2006. ⁵⁵
- [Lu'13] C. Lu, M. Mandal, "Toward automatic mitotic cell detection and segmentation in multispectral histopathological images", in *IEEE Journal of Biomedical and Health Informatics*, vol. 18, no. 2, pp. 594–605, 2013. ³¹
- [Lu'14] G. Lu, B. Fei, "Medical hyperspectral imaging: a review", in *Journal of biomedical optics*, vol. 19, no. 1, p. 010901, 2014. ^{31, 36}
- [Lu'17] G. Lu, J. V. Little, X. Wang, H. Zhang, M. R. Patel, C. C. Griffith, M. W. El-Deiry, A. Y. Chen, B. Fei, "Detection of head and neck cancer in surgical specimens using quantitative hyperspectral imaging", in *Clinical Cancer Research*, vol. 23, no. 18, pp. 5426–5436, 2017. ³⁴
- [Ma'14] L. Ma, M. M. Crawford, X. Yang, Y. Guo, "Local-manifold-learning-based graph construction for semisupervised hyperspectral image classification", in *IEEE Transactions on Geoscience and Remote Sensing*, vol. 53, no. 5, pp. 2832–2844, 2014. ¹²⁰
- [Madroñal'16a] D. Madroñal, H. Fabelo, R. Lazcano, G. Callicó, E. Juárez, C. Sanz, "Parallel implementation of a hyperspectral image linear SVM classifier using RVC-CAL", in *High-Performance Computing in Geoscience and Remote Sensing VI*, vol. 10007, p. 1000709, International Society for Optics and Photonics, 2016. ²¹⁷

BIBLIOGRAPHY

- [Madroñal'16b] D. Madroñal, R. Lazcano, H. Fabelo, S. Ortega, G. M. Callicó, E. Juárez, C. Sanz, "Hyperspectral image classification using a parallel implementation of the linear SVM on a Massively Parallel Processor Array (MPPA) platform", in *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 154–160, IEEE, 2016. ²¹⁶
- [Madroñal'17a] D. Madroñal, R. Lazcano, H. Fabelo, S. Ortega, R. Salvador, G. M. Callicó, E. Juárez, C. Sanz, "Energy consumption characterization of a Massively Parallel Processor Array (MPPA) platform running a hyperspectral SVM classifier", in *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1–6, IEEE, 2017. ²¹⁵
- [Madroñal'17b] D. Madroñal, R. Lazcano, R. Salvador, H. Fabelo, S. Ortega, G. M. Callicó, E. Juárez, C. Sanz, "SVM-based real-time hyperspectral image classifier on a manycore architecture", in *Journal of Systems Architecture*, vol. 80, pp. 30–40, 2017. ^{35, 119, 176, 181, 214}
- [Madroñal'18] D. Madroñal, A. Morvan, R. Lazcano, R. Salvador, K. Desnos, E. Juárez, C. Sanz, "Automatic instrumentation of dataflow applications using papi", in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pp. 232–235, ACM, 2018. ²¹⁵
- [Madronal'19] D. Madronal, F. Arrestier, J. Sancho, A. Morvan, R. Lazcano, K. Desnos, R. Salvador, D. Menard, E. Juárez, C. Sanz, "PAPIFY: Automatic Instrumentation and Monitoring of Dynamic Dataflow Applications Based on PAPI", in *IEEE Access*, vol. 7, pp. 111 801–111 812, 2019. ^{58, 110, 213}
- [Makantasis'15] K. Makantasis, K. Karantzalos, A. Doulamis, N. Doulamis, "Deep supervised learning for hyperspectral data classification through convolutional neural networks", in *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pp. 4959–4962, IEEE, 2015. ³⁷
- [Manolakis'02] D. Manolakis, G. Shaw, "Detection algorithms for hyperspectral imaging applications", in *IEEE signal processing magazine*, vol. 19, no. 1, pp. 29–43, 2002. ³⁰
- [Martel'18] E. Martel, R. Lazcano, J. López, D. Madroñal, R. Salvador, S. López, E. Juárez, R. Guerra, C. Sanz, R. Sarmiento, "Implementation of the principal component analysis onto high-performance computer facilities for hyperspectral dimensionality reduction: Results and comparisons", in *Remote Sensing*, vol. 10, no. 6, p. 864, 2018. ^{176, 213}
- [Masood'06] K. Masood, N. Rajpoot, K. Rajpoot, H. Qureshi, "Hyperspectral colon tissue classification using morphological analysis", in *2006 International Conference on Emerging Technologies*, pp. 735–741, IEEE, 2006. ³⁷
- [Meer'12] F. D. Van der Meer, H. M. Van der Werff, F. J. Van Ruitenbeek, C. A. Hecker, W. H. Bakker, M. F. Noomen, M. Van Der Meijde, E. J. M. Carranza, J. B. De Smeth, T. Woldai, "Multi-and hyperspectral geologic remote sensing: A review", in *International Journal of Applied Earth Observation and Geoinformation*, vol. 14, no. 1, pp. 112–128, 2012. ³¹

-
- [Meijer'10] S. Meijer, H. Nikolov, T. Stefanov, "Throughput modeling to evaluate process merging transformations in polyhedral process networks", in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 747–752, IEEE, 2010. ²⁰
- [Meister'09] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, R. Lethin, "Productivity via automatic code generation for pgas platforms with the r-stream compiler", in *Workshop on Asynchrony in the PGAS Programming Model*, 2009. ²³
- [Melgani'04] F. Melgani, L. Bruzzone, "Classification of hyperspectral remote sensing images with support vector machines", in *IEEE Transactions on geoscience and remote sensing*, vol. 42, no. 8, pp. 1778–1790, 2004. ¹¹⁹
- [Michie'68] D. Michie, "'Memo' Functions and Machine Learning", in *Nature*, vol. 218, no. 306, Apr. 1968, doi:10.1038/218019a0, <https://doi.org/10.1038/218019a0>. ⁷⁹
- [Milanic'15] M. Milanic, A. Bjorgan, M. Larsson, T. Strömberg, L. L. Randeberg, "Detection of hypercholesterolemia using hyperspectral imaging of human skin", in *European Conference on Biomedical Optics*, p. 95370C, Optical Society of America, 2015. ³¹
- [Mishra'11] B. S. P. Mishra, S. Dehuri, "Parallel computing environments: A review", in *IETE Technical Review*, vol. 28, no. 3, pp. 240–247, 2011. ¹⁵
- [moc'18] "CERBERO Deliverable D3.5 - CERBERO Models of Computation", 2018, <https://www.cerbero-h2020.eu/wp-content/uploads/2018/12/D3.5.pdf>. ¹⁰
- [Molony'17] P. Molony, N. Kharytaniuk, S. Boyle, R. S. Woods, G. O'Leary, R. Werner, C. Heffron, L. Feeley, P. Sheahan, "Impact of positive margins on outcomes of oropharyngeal squamous cell carcinoma according to p16 status", in *Head & neck*, vol. 39, no. 8, pp. 1680–1688, 2017. ³⁴
- [Mondal'17] M. A. M. Mondal, M. H. Ali, "Performance Review of the Stereo Matching Algorithms", in *American Journal of Computer Science and Information Engineering*, vol. 4, no. 1, pp. 7–15, 2017. ¹¹⁰
- [Navarro'14] C. A. Navarro, N. Hitschfeld-Kahler, L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using GPU architectures", in *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014. ¹⁵
- [Oancea'09] C. E. Oancea, A. Mycroft, T. Harris, "A Lightweight In-place Implementation for Software Thread-level Speculation", in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pp. 223–232, ACM, New York, NY, USA, 2009, ISBN 978-1-60558-606-9, doi:10.1145/1583991.1584050, <http://doi.acm.org/10.1145/1583991.1584050>. ⁵⁵

BIBLIOGRAPHY

- [Orditura'14] M. Orditura, G. Galizia, V. Sforza, V. Gambardella, A. Fabozzi, M. M. Laterza, F. Andreozzi, J. Ventriglia, B. Savastano, A. Mabilia, et al., "Treatment of gastric cancer", in *World journal of gastroenterology: WJG*, vol. 20, no. 7, p. 1635, 2014. ³²
- [Ortega'17] S. Ortega, H. Fabelo, R. Camacho, M. Plaza, G. Callico, R. Lazcano, D. Madroñal, R. Salvador, E. Juárez, R. Sarmiento, "P03. 18 Detection of human brain cancer in pathological slides using hyperspectral images", in *Neuro-oncology*, vol. 19, no. Suppl 3, p. iii37, 2017. ²¹⁴
- [Ortega'18] S. Ortega, H. Fabelo, R. Camacho, M. De la Luz Plaza, G. M. Callicó, R. Sarmiento, "Detecting brain tumor in pathological slides using hyperspectral imaging", in *Biomedical optics express*, vol. 9, no. 2, pp. 818–831, 2018. ³¹
- [Ortega'19] S. Ortega, H. Fabelo, D. K. Iakovidis, A. Koulaouzidis, G. M. Callico, "Use of hyperspectral/multispectral imaging in gastroenterology. Shedding some–different–light into the dark", in *Journal of clinical medicine*, vol. 8, no. 1, p. 36, 2019. ³¹
- [Ostroff'95] J. S. Ostroff, "Abstraction and composition of discrete real-time systems", in *Proc. of CASE*, vol. 95, no. 370-380, p. 17, 1995. ¹¹
- [Ottermo'06] M. V. Ottermo, M. Øvstedal, T. Langø, Ø. Stavadahl, Y. Yavuz, T. A. Johansen, R. Mårvik, "The role of tactile feedback in laparoscopic surgery", in *Surgical Laparoscopy Endoscopy & Percutaneous Techniques*, vol. 16, no. 6, pp. 390–400, 2006. ³²
- [Palumbo'19a] F. Palumbo, T. Fanni, C. Sau, L. Pulina, L. Raffo, M. Masin, E. Shindin, P. S. de Rojas, K. Desnos, M. Pelcat, et al., "CERBERO: Cross-layer model-based framework for multi-objective design of reconfigurable systems in uncertain hybrid environments: Invited paper: CERBERO teams from UniSS, UniCA, IBM Research, TASE, INSA-Rennes, UPM, USI, Abinsula, AmbieSense, TNO, S&T, CRF", in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pp. 320–325, ACM, 2019. ^{1, 40}
- [Palumbo'19b] F. Palumbo, T. Fanni, C. Sau, A. Rodríguez, D. Madroñal, K. Desnos, A. Morvan, M. Pelcat, C. Rubattu, R. Lazcano, et al., "Hardware/Software Self-adaptation in CPS: The CERBERO Project Approach", in *International Conference on Embedded Computer Systems*, pp. 416–428, Springer, 2019. ²¹⁵
- [Panasyuk'07] S. V. Panasyuk, S. Yang, D. V. Faller, D. Ngo, R. A. Lew, J. E. Freeman, A. E. Rogers, "Medical hyperspectral imaging to facilitate residual tumor identification during surgery", in *Cancer biology & therapy*, vol. 6, no. 3, pp. 439–446, 2007. ³³
- [Pelcat'09] M. Pelcat, J. F. Nezan, J. Piat, J. Croizer, S. Aridhi, "A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems", 2009. ⁵⁸

- [Pelcat'14] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming", in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pp. 36–40, Sept 2014, doi: 10.1109/EDERC.2014.6924354. ^{19, 56, 110}
- [Poletto'97] M. Poletto, D. R. Engler, M. F. Kaashoek, "Tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation", in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI '97*, pp. 109–121, ACM, New York, NY, USA, 1997, ISBN 0-89791-907-6, doi:10.1145/258915.258926, <http://doi.acm.org/10.1145/258915.258926>. ⁸⁴
- [Pouchet'12] L.-N. Pouchet, et al., "Polybench: The polyhedral benchmark suite", in *URL: <http://www.cs.ucla.edu/pouchet/software/polybench>*, 2012. ²⁸
- [Pourreza-Shahri'13] R. Pourreza-Shahri, F. Saki, N. Kehtarnavaz, P. Leboulluec, H. Liu, "Classification of ex-vivo breast cancer positive margins measured by hyperspectral imaging", in *2013 IEEE international conference on image processing*, pp. 1408–1412, IEEE, 2013. ³³
- [Probst'02] M. Probst, "Dynamic binary translation", in *UKUUG Linux Developer's Conference*, vol. 2002, 2002. ²⁵
- [Raman'10] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, D. I. August, "Speculative Parallelization Using Software Multi-threaded Transactions", in *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 65–76, Mar. 2010, ISSN 0163-5964, doi:10.1145/1735970.1736030, <http://doi.acm.org/10.1145/1735970.1736030>. ⁵⁵
- [Ravi'16] D. Ravi, C. Wong, F. Deligianni, M. Berthelot, J. Andreu-Perez, B. Lo, G.-Z. Yang, "Deep learning for health informatics", in *IEEE journal of biomedical and health informatics*, vol. 21, no. 1, pp. 4–21, 2016. ³⁷
- [Regeling'16a] B. Regeling, W. Laffers, A. O. Gerstner, S. Westermann, N. A. Müller, K. Schmidt, J. Bendix, B. Thies, "Development of an image pre-processor for operational hyperspectral laryngeal cancer detection", in *Journal of biophotonics*, vol. 9, no. 3, pp. 235–245, 2016. ³¹
- [Regeling'16b] B. Regeling, B. Thies, A. O. Gerstner, S. Westermann, N. A. Müller, J. Bendix, W. Laffers, "Hyperspectral imaging using flexible endoscopy for laryngeal cancer detection", in *Sensors*, vol. 16, no. 8, p. 1288, 2016. ^{34, 37}
- [Salim'15] M. Salim, A. O. Akkirman, M. Hidayetoglu, L. Gurel, "Comparative benchmarking: Matrix multiplication on a multicore coprocessor and a GPU", in *2015 Computational Electromagnetics International Workshop (CEM)*, pp. 1–2, IEEE, 2015. ¹¹⁰
- [Salvador'16] R. Salvador, H. Fabelo, R. Lazcano, S. Ortega, D. Madroñal, G. M. Callicó, E. Juárez, C. Sanz, "HELICoiD tool demonstrator for real-time brain

BIBLIOGRAPHY

- cancer detection”, in *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 237–238, IEEE, 2016. ^{217, 221}
- [Salvador’17] R. Salvador, S. Ortega, D. Madroñal, H. Fabelo, R. Lazcano, G. Marrero, E. Juárez, R. Sarmiento, C. Sanz, “HELICoiD: Interdisciplinary and collaborative project for real-time brain cancer detection”, in *Proceedings of the Computing Frontiers Conference*, pp. 313–318, ACM, 2017. ^{35, 114, 116, 118, 216}
- [Sanai’08] N. Sanai, M. S. Berger, “Glioma extent of resection and its impact on patient outcome”, in *Neurosurgery*, vol. 62, no. 4, pp. 753–766, 2008. ³⁴
- [Sanderson’16] C. Sanderson, R. Curtin, “Armadillo: a template-based C++ library for linear algebra”, in *Journal of Open Source Software*, vol. 1, no. 2, p. 26, 2016. ⁷⁸
- [Sato’17] Y. Sato, T. Yuki, T. Endo, “ExanaDBT: A dynamic compilation system for transparent polyhedral optimizations at runtime”, in *Proceedings of the Computing Frontiers Conference*, pp. 191–200, ACM, 2017. ²⁵
- [Savage’98] J. E. Savage, “Models of computation. Vol. 136”, 1998. ²
- [Shen’13] J. P. Shen, M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*, Waveland Press, 2013. ^{20, 52}
- [Sobel’68] I. Sobel, G. Feldman, “A 3x3 isotropic gradient operator for image processing”, in *a talk at the Stanford Artificial Project in*, pp. 271–272, 1968. ^{57, 110}
- [Sorel’05] Y. Sorel, “From modeling/simulation with Scilab/Scicos to optimized distributed embedded real-time implementation with SynDEx”, in *Proceedings of the International Workshop On Scilab and Open Source Software Engineering, SOSSE’05*, Wuhan, China, Oct. 2005, <http://www-rocq.inria.fr/syndex/publications/pubs/sosse05/sosse05.pdf>. ¹⁹
- [Spek’08] H. L. van der Spek, E. M. Bakker, H. A. Wijshoff, “SPARK00: A benchmark package for the compiler evaluation of irregular/sparse codes”, in *arXiv preprint arXiv:0805.3897*, 2008. ^{29, 113}
- [Sriram’18] S. Sriram, S. S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*, CRC press, 2018. ⁶⁰
- [Starr’10] C. Starr, C. Evers, L. Starr, *Biology: concepts and applications without physiology*, Cengage Learning, 2010. ³⁰
- [Steffan’98] J. G. Steffan, T. C. Mowry, “The potential for using thread-level data speculation to facilitate automatic parallelization”, in *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, pp. 2–13, IEEE, 1998. ^{22, 51, 68}

-
- [Steffan'05] J. G. Steffan, C. Colohan, A. Zhai, T. C. Mowry, "The STAMPede approach to thread-level speculation", in *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 3, pp. 253–300, 2005. ⁵⁵
- [Stone'10] J. E. Stone, D. Gohara, G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems", in *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010. ¹⁷
- [Stratton'12] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing", in *Center for Reliable and High-Performance Computing*, vol. 127, 2012. ^{29, 111}
- [Stripf'13] T. Stripf, O. Oey, T. Bruckschloegl, J. Becker, G. Rauwerda, K. Sunesen, G. Goulas, P. Alefragis, N. S. Voros, S. Derrien, et al., "Compiling Scilab to high performance embedded multicore systems", in *Microprocessors and Microsystems*, vol. 37, no. 8, pp. 1033–1049, 2013. ²⁴
- [Stummer'06] W. Stummer, U. Pichlmeier, T. Meinel, O. D. Wiestler, F. Zanella, H.-J. Reulen, A.-G. S. Group, et al., "Fluorescence-guided surgery with 5-aminolevulinic acid for resection of malignant glioma: a randomised controlled multicentre phase III trial", in *The lancet oncology*, vol. 7, no. 5, pp. 392–401, 2006. ³⁵
- [Stummer'11] W. Stummer, J.-C. Tonn, H. M. Mehdorn, U. Nestler, K. Franz, C. Goetz, A. Bink, U. Pichlmeier, "Counterbalancing risks and gains from extended resections in malignant glioma surgery: a supplemental analysis from the randomized 5-aminolevulinic acid glioma resection study", in *Journal of neurosurgery*, vol. 114, no. 3, pp. 613–623, 2011. ³⁴
- [Szolna'16] A. Szolna, J. Morera, J. Piñeiro, G. Callicó, H. Fabelo, S. Ortega, "Hyperspectral Imaging as A Novel Instrument for Intraoperative Brain Tumor Detection", in *Neurocirugia*, vol. 27, p. 166, 2016. ³⁵
- [Tabar'16] Y. R. Tabar, U. Halici, "A novel deep learning approach for classification of EEG motor imagery signals", in *Journal of neural engineering*, vol. 14, no. 1, p. 016 003, 2016. ³⁷
- [Tarabalka'09] Y. Tarabalka, J. A. Benediktsson, J. Chanussot, "Spectral-spatial classification of hyperspectral imagery based on partitional clustering techniques", in *IEEE Transactions on Geoscience and Remote Sensing*, vol. 47, no. 8, pp. 2973–2987, 2009. ¹¹⁷
- [Trifunovic'10] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, R. Upadrasta, "Graphite two years after: First lessons learned from real-world polyhedral compilation", 2010. ²³
- [Valiant'90] L. G. Valiant, "A bridging model for parallel computation", in *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990. ¹²

BIBLIOGRAPHY

- [Verdoolaege'07a] S. Verdoolaege, H. Nikolov, T. Stefanov, "PN: a tool for improved derivation of process networks", in *EURASIP journal on Embedded Systems*, vol. 2007, no. 1, p. 075 947, 2007. ²⁰
- [Verdoolaege'07b] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, M. Bruynooghe, "Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions", in *Algorithmica*, vol. 48, no. 1, pp. 37–66, 2007. ⁷⁵
- [Verdoolaege'10] S. Verdoolaege, "isl: An integer set library for the polyhedral model", in *International Congress on Mathematical Software*, pp. 299–302, Springer, 2010. ²⁴
- [Verdoolaege'13a] S. Verdoolaege, "Polyhedral process networks", in *Handbook of Signal Processing Systems*, pp. 1335–1375, Springer, 2013. ¹³
- [Verdoolaege'13b] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, E. Catthoor, "Polyhedral parallel code generation for CUDA", in *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 54, 2013. ²⁴
- [Vigneswaran'14] N. Vigneswaran, M. D. Williams, "Epidemiologic trends in head and neck cancer and aids in diagnosis", in *Oral and Maxillofacial Surgery Clinics*, vol. 26, no. 2, pp. 123–141, 2014. ³⁴
- [Volkov'08] V. Volkov, J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra", in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–11, IEEE, 2008. ¹¹⁰
- [Wang'16] S. Wang, M. Yang, S. Du, J. Yang, B. Liu, J. M. Gorriz, J. Ramírez, T.-F. Yuan, Y. Zhang, "Wavelet entropy and directed acyclic graph support vector machine for detection of patients with unilateral hearing loss in MRI scanning", in *Frontiers in computational neuroscience*, vol. 10, p. 106, 2016. ³⁶
- [Wu'04] T.-F. Wu, C.-J. Lin, R. C. Weng, "Probability estimates for multi-class classification by pairwise coupling", in *Journal of Machine Learning Research*, vol. 5, no. Aug, pp. 975–1005, 2004. ^{119, 180, 181}
- [Wu'13] D. Wu, D.-W. Sun, "Advanced applications of hyperspectral imaging technology for food quality and safety analysis and assessment: A review—Part I: Fundamentals", in *Innovative Food Science & Emerging Technologies*, vol. 19, pp. 1–14, 2013. ³¹
- [Yao'07] M. Yao, J. B. Epstein, B. J. Modi, K. B. Pytynia, A. J. Mundt, L. E. Feldman, "Current surgical treatment of squamous cell carcinoma of the head and neck", in *Oral oncology*, vol. 43, no. 3, pp. 213–223, 2007. ³⁴
- [Young'15] R. M. Young, A. Jamshidi, G. Davis, J. H. Sherman, "Current trends in the surgical management and treatment of adult glioblastoma", in *Annals of translational medicine*, vol. 3, no. 9, 2015. ³⁵

- [Yuen'10] P. W. Yuen, M. Richardson, "An introduction to hyperspectral imaging and its application for security, surveillance and target acquisition", in *The Imaging Science Journal*, vol. 58, no. 5, pp. 241–253, 2010. ³¹
- [Yuki'12] T. Yuki, G. Gupta, D. Kim, T. Pathan, S. Rajopadhye, "Alphaz: A system for design space exploration in the polyhedral model", in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 17–31, Springer, 2012. ²³
- [Yuki'13] T. Yuki, A. Morvan, S. Derrien, "Derivation of efficient FSM from loop nests", in *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 286–293, IEEE, 2013. ^{21, 48}
- [Yviquel'13] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, M. Raulet, "Orcc: Multimedia development made easy", in *Proceedings of the 21st ACM international conference on Multimedia*, pp. 863–866, ACM, 2013. ¹⁹
- [Zhang'16a] L. Zhang, L. Zhang, B. Du, "Deep learning for remote sensing data: A technical tutorial on the state of the art", in *IEEE Geoscience and Remote Sensing Magazine*, vol. 4, no. 2, pp. 22–40, 2016. ³⁷
- [Zhang'16b] Y. Zhang, S. Lu, X. Zhou, M. Yang, L. Wu, B. Liu, P. Phillips, S. Wang, "Comparison of machine learning methods for stationary wavelet entropy-based multiple sclerosis detection: decision tree, k-nearest neighbors, and support vector machine", in *Simulation*, vol. 92, no. 9, pp. 861–871, 2016. ³⁶