The CIAO Multi-Dialect Compiler and System: A Demo and a Status Report

M. Hermenegildo† F. Bueno† D. Cabeza† M. Carro†
M. García de la Banda‡ P. López‡ G. Puebla‡

Extended Abstract

This paper presents a brief overview of the CIAO system, the capabilities of its compiler and the techniques used for supporting them. It essentially provides an overview of current work at the UPM CLIP group, performed in collaboration with the U. of Arizona, K. U. Leuven, Melbourne U., Monash U., and New Mexico State U. More details and references on this project can be found in the full version of the paper [Her].

CIAO is a compiler, run-time, and program development system which supports several programming paradigms and execution models in a unified framework. It was initially conceived as a compiler writer’s workbench, i.e., a testbed for advanced compilation techniques, but has since proven also quite useful for application development. Programming styles supported include those of Prolog/Logic Programming, CLP, and CC. In the belief that network applications are a good target for computational logic systems, the styles above are combined in CIAO with distributed and network-wide execution capabilities. Such capabilities allow both the transparent execution of parallel and concurrent code written for a multiprocessor in a distributed environment [CH95] and the straightforward generation of WWW-based applications [CH96a, CH96b, CHV96]. CIAO also supports several computation rules, including standard left-to-right SLD resolution and the determinate-first principle (as in the Andorra model [SCWY90, dMSC93]).

The implementation of CIAO is based on the observation that, under certain assumptions, a large number of currently proposed LP, CLP, and CC programming and execution models can be explained and implemented through the application of only a few basic principles and constructs, and that there is much in common at the abstract machine level among such models [HtCg93, HtCg94]. Thus, CIAO programs are compiled and optimized via source to source transformations into a kernel language (which is also the native CIAO language). We believe medium to high performance implementations can be obtained in

---

†This work has been funded in part by ESPRIT projects PRINCE, PARFORCE, and ACCLAIM, and CICYT project IPL-D.
‡CS Dept., Technical U. of Madrid. {herme,bueno,dcabeza,mcarro,pig,german}@fi.upm.es
§Dept. of CS, Monash University, Clayton 3168, Australia. mbanda@bruce.cs.monash.oz.au
this way, with the advantages that most analysis phases can be performed at the kernel language level and the same analyzer can be shared among several execution models. Also, optimizations can be performed via source to source transformations, and the low-level machinery can be kept minimal. Optimizations in CIAO include multivariant (abstract) specialization and automatic program parallelization [CC94]. The kernel language in turn is directly supported by a comparatively simple, generic abstract machine including parallelism, concurrency, and distributed execution capabilities (and which is essentially an extension of the &-Prolog abstract machine [Her86, HG91]). It includes native support for attributed variables [Hol92, Hu90, Neu90], which are used in the implementation of constraint solvers (as in Eclipse [Eur93] and SICStus 3 [Swe95]) and for communication among concurrent tasks [HCC95]. In addition, the CIAO compiler can also support a significant part of the system’s functionality by compilation into certain Prolog dialects (those which include coroutining and attributed variables, such as Eclipse or SICStus 3). Thus, the CIAO system can also be seen as a library or front end for current Prolog systems.

One of the design decisions in CIAO is to provide explicit control capabilities in the kernel language. This allows the system to perform optimizations such as parallelization [BGH94] (task creation based on dependencies), partitioning and schedule analysis (task coalescence based on dependencies), and granularity control (task coalescence based on task size considerations) as source to source transformations. Such control is provided first by separate sequentiality (“”), concurrency (“&/1”), and parallelism (“&/2”, “&>/2”, “<&/1”) operators. The parallel operators allow indicating points where parallelism can be exploited. Their behavior is otherwise equivalent to that of the sequential operator (full backtracking is supported). These operators essentially assume independence among goals and communication of bindings is therefore not guaranteed until the join. No variable locking is performed. The concurrency operator allows concurrent programming in the style of CC languages (also, the parallelism in such concurrent execution may be exploited if resources are available). Backtracking is limited to allow a relatively straightforward implementation. Variable communication (and locking) is performed. In addition, a fair concurrency operator is provided (“&&/1”) which explicitly requests the (efficient) association of computational resources (e.g., an operating system thread) to a goal. This allows implementing a fair source language that compiles efficiently into this and the above operators (perhaps via an analysis which can determine the program points where fairness is really needed – to ensure, for example, termination). Also, explicit synchronization is provided in the kernel language by means of “wait/1” and “ask/1” operators (the latter as in &-Prolog), augmented with some meta-tests on the variables (the latter, using the attributed variables approach of Holzbaur [Hol90]). Finally, a placement operator (“@”) allows control of task placement in distributed execution. Other primitives for controlling distributed execution deal with teams of workers, and with using active modules or active objects.

The CIAO compiler first performs a translation of the input source into the kernel

---

1While the abstract machine currently supports only (“dependent” and “independent”) and-parallelism, its combination with or-parallelism is planned by applying the techniques developed in ACE and similar models [GSCYH91, GC92, GHPC94].
language, followed by (incremental) global analysis and optimization phases. The resulting program is finally compiled into abstract machine code (including byte-code files, stand-alone executables, and incore compilation). Alternatively, the compiler can produce output which can run on a Prolog system supporting delay and attributed variables, as mentioned before. It is also possible to obtain the results of each of the intermediate compilation phases. This allows visualizing and affecting the transformation, analysis, parallelization, and optimization steps. Because of the source to source nature of the compiler, this output is always a (possibly annotated) kernel CIAO program. Also, (parallel) execution traces can be generated which can be analyzed by several graphical performance analysis tools.

The compiler currently supports the CIAO kernel language (backwards compatible with Prolog, but extended with the specific CIAO primitives mentioned before), languages based on the basic Andorra model, and basic CC languages. Program transformations bridge the semantic gaps between the different programming paradigms supported. The methods used for translating programs based on the (Basic) Andorra model to CIAO are described in [BDGH95]. The methods used for translating CC languages are an extension of those of [DBG94, Deb93] and are described in [BH95b]. Also, for each of these languages the system currently supports the constraint domains of Prolog, CLP(R), and CLP(Q). Finally, there is support for programming in the functional style, both in terms of syntax (allowing functional notation and nesting of calls via a designated output argument in relations) and of improved support at the abstract machine for meta-programming (higher-order).

The CIAO compiler includes both local and global analysis of programs. Global analysis is performed in the context of abstract interpretation [CC77, Deb92, CC92]. The underlying framework of analysis is that of PLAI [HWD92, MH90, MH92], which implements a generic (goal-dependent and goal-independent) top-down driven abstract interpreter. The genericity of the framework allows plugging in different abstract domains, provided suitable interfacing functions are defined. PLAI also incorporates incremental analysis [HMPS95] in order to deal with large programs and is capable of analyzing full languages (in particular, full standard Prolog [BCHP96, CRH94]). A modification of the PLAI framework capable of analyzing dynamically scheduled programs is also included in order to support the concurrent models. Note that, thanks to the transformational approach, the same framework allows analyzing programs with delays [Col82, Car87] and CC languages with angelic nondeterminism. Initial studies showed that accurate analysis in such programs is possible [MGH94], although this technique involves relatively large cost in analysis time. The analysis integrated into the CIAO compiler, developed in collaboration with Monash and Melbourne U., uses a novel method based on approximating the delayed atoms by a closure operator, which improves efficiency without significant loss of accuracy [GMS95].

The CIAO analyzer currently includes several domains, some of which have been implemented by other users of the PLAI system, notably the K. U. Leuven, Monash U., and Melbourne U. For the analysis of (classical) logic programs a number of traditional domains capturing information on variable groundness, freeness, sharing, and linearity are included. These include the set sharing $Sh$ [JL89, MH89], set sharing and freeness $Sh+Fr$ [MH91], and

---

2This is a kind of nondeterminism which does not give rise to an arbitrary choice when applying a search rule.
pair sharing $\textit{ASub}$ [Son86] domains. Combinations of $\textit{Sh}$ and $\textit{Sh+Fr}$ with $\textit{ASub}$ are also supported, resulting in the $\textit{Sh+ASub}$ and $\textit{Sh+Fr+ASub}$ domains. The combination is done in such a way that the original domains and operations of the analyzer over them are reused, instead of redefining the domains for the combination [CC79, CMB+95]. Three other domains, modified versions of $\textit{Path sharing}$, $\textit{Aeqns}$ (abstract equations), and $\textit{Types}$ have recently also been incorporated (the latter is needed in granularity control and also allows interactive type checking and generation). Additional domains are supported for CLP. The abstract domain $\textit{Def}$ [GH93, Gar94] determines whether program variables are definite, i.e. constrained to a unique value. In doing this it keeps track of $\textit{definite}$ dependencies among variables. The abstract domain $\textit{Fr}$ [DJBC93, DJ94, Dum94] determines which variables act as $\textit{degrees of freedom}$ with respect to the satisfiability of the constraint store in which they occur. In doing this it keeps track of $\textit{possible}$ dependencies among variables. A combined domain $\textit{Fd}$ which infers both definiteness and freeness is also integrated. Finally, a preliminary version of the domain $\textit{LSign}$ [MS94] is also supported.

The compile-time $\textit{parallelization}$ module currently aims at uncovering goal-level, restricted (i.e., fork and join), independent and-parallelism (independence has the very desirable properties of correct and efficient execution w.r.t. standard sequential execution of Prolog or CLP). In the context of LP, parallelization is performed based on the well-understood concepts of $\textit{strict}$ and $\textit{non-strict}$ independence [HR95], using the information provided by the abstract domains. Also, appropriate parallelizers using recently proposed definitions of independence for CLP [GHM93, Gar94] (and for constraint programming with dynamic scheduling) have been implemented in order to parallelize CLP and CC programs [GHM95, GBH96], making use of the information from the CLP analyses.

The optimization phase in the compiler is strongly based on (abstract) program specialization. Literals and predicates which are proved to always succeed, fail, be reducible to a few unifications, or lead to error are simplified or eliminated. The same principle applies at lower levels. This can speed up the program at run-time (by eliminating static code such as run-time tests that are always known to succeed), and also be useful to detect programming errors at compile-time. At user request the compiler can also specialize the program using the versions generated during analysis [PH95a]. This may involve generating different versions of a predicate for different $\textit{abstract call patterns}$, thus increasing the program size. In order to keep the size of the specialized program as reduced as possible, the number of versions of each predicate is minimized to those which are found useful for performing optimizations. The optimization module also supports optimization in the context of dynamic scheduling [PH95b]. The optimizations then include simplification and elimination of suspension conditions and elimination of concurrency primitives (sequentialization).

As well as handling sequential code, the optimization module of the CIAO compiler contains what we believe is the first automatic optimizer for languages with dynamic scheduling [PH95b]. The potential benefits of the optimization of this type of programs were already shown in [MGH94], but they can now be obtained automatically. These kinds of optimizations include simplification and elimination of suspension conditions and elimination of concurrency primitives (sequentialization).

A further step in the compiler provides granularity control for parallel execution using
the techniques described in [DLH90, Tic88, LHD94, DGHL94, LH95]. The compiler estimates the granularity of parallel tasks, i.e. the work available under them, by generating expressions that are upper and lower bounds for the computation time of parallel tasks as a function of the size of task input data. These functions are used at run-time to perform granularity control by comparing cost bounds to suitable thresholds. The preliminary results obtained are quite encouraging, although there remains much work to be done in this very important area.

The back end of the compiler takes the result of the previous program transformations and generates a number of final output formats. Normally, the result of the compiler is intended for the CIAO/&-Prolog abstract machine. Output possibilities are then byte-code (".qi") files, stand-alone executables, and incore compilation (when the compiler is running inside the system rather than as a stand-alone application). As mentioned before, and as an alternative output, most of the capability of the system can also be handled by any Prolog which supports delay declarations and attributed variables. Alternatively, also AKL [JH91] can be used as a target, using the techniques described in [BH95a].

Two tools have been developed to complement the environment and help during performance debugging. VisAndOr [CGH93] is a tool for visualizing parallel executions. It supports both conjunctive and disjunctive execution graphs (or- and and-parallelism). It is currently in use by several other researchers in the field and it is distributed with other parallel systems such as Muse [AK90]. IDRA [FCH96] is a simulator which can quite accurately compute ideal speedups from traces from a sequential execution of a parallelized program. It allows evaluating the performance of the parallel run-time system independently of the quality of the parallelization performed by the compiler, by comparing the obtained speedups with those predicted by IDRA for the given parallelized program.

Current versions of different parts of the &-Prolog and CIAO systems are publicly available for experimentation (please contact the authors; further information can be obtained from http://www.clip.dia.fi.upm.es/).

References


[BGH94] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Paralleliza-


[GHM95] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence and Search Space Preservation in Dynamically Scheduled Constraint Logic


