A New Approach to Memory Partitioning in On-board Spacecraft Software*

Santiago Urueña, José A. Pulido, Jorge López,
Juan Zamorano, and Juan A. de la Puente

Universidad Politénica de Madrid (UPM), E28040 Madrid, Spain
{suruena,pulido,jorgel,jzamorano,jpuente}@dit.upm.es
http://www.dit.upm.es/rts/

Abstract. The current trend to use partitioned architectures in on-board spacecraft software requires applications running on the same computer platform to be isolated from each other both in the temporal and memory domains. Memory isolation techniques currently used in Integrated Modular Avionics for Aeronautics usually require a Memory Management Unit (MMU), which is not commonly available in the kind of processors currently used in the Space domain. Two alternative approaches are discussed in the paper, based on some features of Ada and state-of-the art compilation tool-chains. Both approaches provide safe memory partitioning with less overhead than current IMA techniques. Some footprint and performance metrics taken on a prototype implementation of the most flexible approach are included.

Key words: Ravenscar Ada, high-integrity, hard real-time, embedded systems, integrated modular avionics.

1 Introduction

On-board embedded computers play a crucial role in spacecraft, where they perform both platform control functions, such as guidance and navigation control or telemetry and tele-command management, and payload specific functions, such as instrument control and data acquisition. One distinctive characteristic of on-board computer systems is that computational resources are scarce, due to the need to use radiation-hardened hardware chips and also to weight and power consumption constraints. In this kind of systems, the more computational resources on-board the higher energy consumption, which in turn results in more power cells and thus more weight, increasing the total weight and the costs required to launch the spacecraft. Another key aspect of these systems is the presence of high-integrity and hard real-time requirements, which raises the need for a strict verification and validation (V&V) process both at the system and software levels [1].

* This work has been funded in part by the Spanish Ministry of Education, project no. TIC2005-08665-C03-01 (THREAD), and by the IST Programme of the European Commission under project IST-004033 (ASSERT).

Current trends envisage systems with increased functionality and complexity. Such systems are often composed of several applications that may have different levels of criticality. In such a scenario, the most critical applications must be isolated from the less critical ones, so that the integrity of the former is not compromised by failures occurring in the latter. Isolation has often been achieved by using a *federated* approach, i.e. by allocating different applications to different computers. However, the growth in the number of applications and the increasing processing power of embedded computers foster an *integrated* approach, in which several applications may be executed on a single computer platform. In this case, alternate mechanisms must be put in place in order to isolate applications from each other. The common approach is to provide a number of *logical partitions*[1] on each computer platform, in such a way that each partition is allocated a share of processor time, memory space, and other resources. Partitions are thus isolated from each other both in the temporal and spatial domains. Temporal isolation implies that a partition does not use more processor time than allocated, and spatial isolation means that software running in a partition does not read or write into memory space allocated to another partition.

This approach has been successfully implemented in the aeronautics domain by so-called Integrated Modular Avionics (IMA) [2]. While IMA is industrially supported and effectively provides temporal and spatial isolation, its use in spacecraft systems raises some problems due to the need of complex computer boards that call for alternative, more flexible solutions. In this context, Ada 2005 [3] provides a new set of real-time mechanisms that open the way to new approaches to inter-partition isolation. Some strategies for providing temporal isolation using the new Ada execution-time monitoring mechanisms have already been developed by the authors [4], and prototype implementations have been built in the framework of the ASSERT project[2] [5].

This paper presents new research directed at providing spatial isolation based on alternative approaches to current IMA architectures, including features of the Ada language and operating system-level mechanisms. The basic idea behind the proposed strategies is to modify the compilation toolchain to make a better use of the scarce computational resources at run-time. The available hardware memory protection is still used at run-time, but predictability losses due to address translation in MMUs are avoided. The rest of the paper is organized as follows. Section 2 describes the main aspects of the current IMA architectures. Section 3 introduces some alternative approaches to spatial isolation. Section 4 discusses the architecture of real-time kernels with respect to memory protection, while section 5 details a set of changes needed in the compilation tool-chain needed to implement the two new strategies. Finally, section 6 references some related work, and section 7 summarizes the main conclusions of this paper.

---

[1] Notice that the term *partition* is not used here in the sense defined in the ALRM (10.2/2), but as an implementation of *protection* as specified in the DO-178B (2.3.1).

[2] `http://www.assert-project.net/`

2 Integrated Modular Avionics

Integrated Modular Avionics (IMA) is a generic term to describe an architecture where different avionics applications are executed independently on a single CPU. Applications may have different criticality levels [6], and be logically distributed on different partitions of the same processor or over a network of computers connected by a communication link.

In order to support different criticality levels, applications have to be isolated from each other. Otherwise all the code would have to be certified to the highest criticality, an extremely expensive (and probably impossible) burden. To this purpose, each computer node is divided into one or more partitions, each of which is a virtual container for one or more applications with the same level of criticality, which are isolated in the time and memory domain from applications running in other partitions. An important consequence of partitioning is that applications can be updated individually without requiring re-certification of the whole system. Figure 1 shows an example of an IMA system.

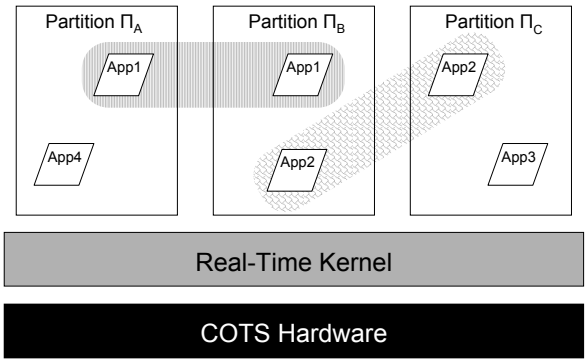

Fig. 1. Four applications with different criticality levels executing inside three partitions over the same computing node.

Implementing an IMA architecture requires a specialized operating system layer that provides temporal and spatial isolation between partitions. The ARINC 653 standard [7] defines an architecture for such an operating system. There are diverse ARINC 653 implementations available from multiple vendors, and the standard has been successfully used in a number of commercial and military avionics systems. However, in spite of its success in the aeronautics field, its application to spacecraft systems raises some problems. First of all, the partition scheduling method is too rigid, and does not allow spare processor time to be re-allocated to other partitions. This may reduce the schedulability of applications on the comparatively slow processors that are currently used in spacecraft computers. The other main problem is that current ARINC 653 implementations

require a MMU, which is seldom available on space computers. Indeed, current processors used by ESA[3], such as LEON2 [8], do not have an MMU. Therefore, other methods not relying on the presence of MMU devices should be explored in order to implement spatial isolation in spacecraft systems.

3 Approaches to spatial isolation

3.1 Static analysis

SPARK is an Ada-based language designed for high-integrity systems. The language is restricted to a safe subset of Ada, augmented with formal annotations enabling efficient static analysis. A particular kind of annotation refers to the integrity—or criticality—level of program elements, enabling static analysis of violations in the criticality segregation [9]. In this way, static information-flow analysis of source code can be used to guarantee that an application will not write into the memory space of another application.

In principle this method can provide spatial isolation for a node with applications with high criticality levels, and it can also be used to ensure fault containment inside a specific application. However, this approach requires all the software in a computer node to be programmed in SPARK, a language intended only for high-criticality applications. Therefore, it is not suitable for the general case where low-criticality applications, possibly written in other languages, are present. On the other hand, it is an interesting approach to spatial isolation in computers which only host highly critical code, and can also be combined with other methods in a more general situation.

3.2 Run-time checks

A second approach is to use the extensive set of compile-time and run-time checks provided by the Ada language to detect possible violations of memory isolation. For example, forbidding using a memory pool in more than one partition seems a reasonable restriction. Following a similar reasoning, the run-time system can be designed so that there is a separate secondary stack for each partition, and an exception is raised in case of overflow. Additional run-time controls for checking that no task can write outside its partition memory area can also be implemented, e.g. when using general access objects a check can be made that the address is inside the partition space, and the same can be done for all access types if 'Unchecked_Access or Unchecked_Conversion is allowed.

Wahbe et al [10] proposed a different software technique to avoid writing outside the memory region of the application called *address sandboxing*. Some code is added before dereferencing a pointer which applies a mask to the high bits of the pointer so that the destination address always falls into the memory range of the application. Therefore, even if the pointer is incorrect, the mask

[3] European Space Agency.

ensures that it will not write outside its memory region. Address sandboxing does not detect failures, but can be more efficient than run-time checks.

The main problem of these approaches is that they add complexity to the compiler and run time support, which may make it difficult to certify high-criticality applications. They can be retained, however, to implement fault containment regions within a partition.

3.3 Hardware protection

Some kind of hardware memory protection is available on virtually all processors, usually allowing read and write access, read-only access, or completely hiding a memory region. In addition, a memory area can be made non-executable, which is useful if the area contains only data. The memory protection setting cannot be modified when the processor is in user mode, but only in supervisor mode, and thus it can only be changed by the operating system. These mechanisms can thus be used to ensure that applications of mixed criticality can safely run on the same node. Furthermore, only the operating system must be certified to the highest criticality level, as it is the only subsystem that deals with memory protection.

An MMU is not always available in spacecraft computers because it is a complex hardware component with a comparatively high power consumption [11], as its internal cache for translating addresses, the TLB, is usually fully-associative and frequently accessed. Moreover, the possibility of TLB misses hinders the predictability of the system and introduces some overhead due to address translation and TLB flushes [12]. The complexity of MMU chips also makes them prone to single event upsets (bit flips due to high-energy particles) [2].

There is another main kind of hardware memory protection mechanism, *fence registers*. Fence registers provide a limited functionality, protecting a fixed number of memory segments of any size. In contrast, an MMU can provide sophisticated memory management schemes, including pagination, segmentation, and virtual memory. While such schemes are usually required in general-purpose operating systems, they are of less use in embedded computing, even with reprogrammability in mind, due to the fact that embedded hard real-time applications are usually statically loaded at system initialization time, at least in spacecrafts. For example, the LEON2 processor has a pair of fence registers that can be used to avoid writings outside the two specified segments of the SRAM.

In this case, there is no hardware relocation, and therefore all applications share a single address space. Memory reads are always allowed by the fence registers. This limits their usefulness as a spatial isolation mechanism, as attempts to read or execute outside the allowed memory area are not detected. In spite of this limitation, fence registers are a simple and robust mechanism without the complexity and comparatively high power consumption of MMUs. Two schemes for implementing spatial isolation based on generic fence registers are described in the following sections.

4 Kernel architecture

4.1 Architecture of current real-time kernels

The current practice in the space domain is to execute all the embedded software in supervisor mode, i.e. any application and not only the kernel can execute privileged instructions. Furthermore, all the code executes inside a single (flat) memory space, and all the applications are linked statically into a single binary image, also including the real-time kernel, regardless of their criticality. As shown in figure 2, all the executable code is linked into a single `.text` section, the global variables are located in the `.data` and `.bss` sections, and the stack for each thread is created in the `.bss` section during initialization.

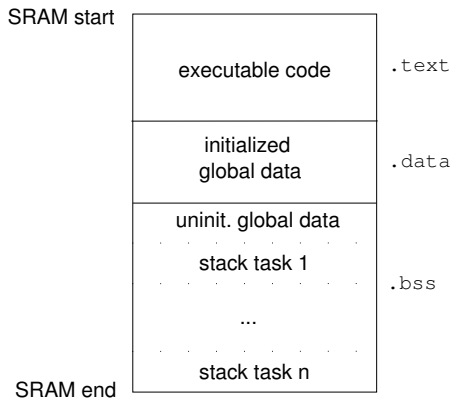

Fig. 2. Current memory map.

This model has several advantages, like increased CPU performance and memory footprint reduction. There is no code duplication because all the applications share the same code, including static libraries. The operating system can be simpler, e.g. there is no application loader. However, hardware memory protection cannot be used to provide complete memory isolation because all the global variables are located in the same section (`.data` or `.bss`), regardless of their criticality level. Only the task stacks can have some memory protection because they are clearly separated in memory. ORK, RTEMS, and ThreadX are examples of real-time kernels currently used in the European space industry that follow this memory allocation model.

4.2 Needed architectural changes

Some changes to the above scheme are required in order to implement spatial isolation using fence registers. Specifically, the global data and stacks (and heap, if available) of each partition must be allocated to separate memory areas, so

that the kernel can provide write permission only to the data area of the partition of the thread that is currently executing.

An example of a memory map implementing this principle is shown in figure 3. In this figure, the code and data of each partition (including the kernel) are grouped into dedicated memory zones. Other schemes are possible, for example one with all the executable code in an adjacent area, which can be more efficient as only one segment has to be used for protecting non-executable memory.

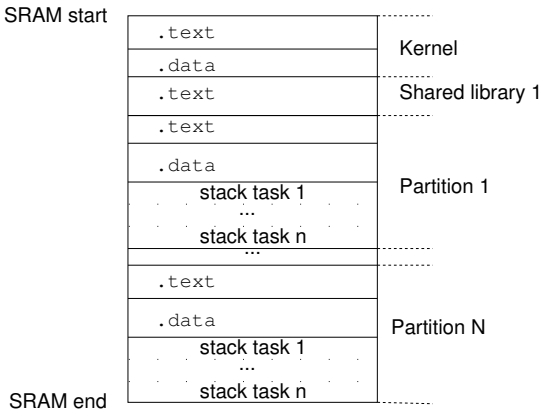

Fig. 3. Memory map for spatial isolation.

It should be noticed that the code shared among partitions is compiled as shared libraries, i.e. each partition using a specific shared library reserves in its private data section the space required for the global variables of the shared library. Otherwise, the code would be duplicated in each partition thus increasing the memory footprint. In addition, it is worth noting that some free memory space should be reserved for on-line reprogrammability.

The above schemes show that implementing spatial isolation with fence register requires changes not only in the real-time kernel, but in the compilation and linking process as well. These changes are discussed in the next section.

5 Modifying the compilation toolchain

5.1 Basic considerations

In order to implement a partitioned system, the tasks and global data that are included in each partition must be identified in the first place. Possible communication between co-operating applications running on different partitions must also be analysed. This in turn requires some kind of inter-partition communication mechanism to be defined.

In the following paragraphs two alternative strategies for developing partitioned systems are explored. The first one is based on building a custom *linking*

script for the partitioned system, and the second one uses a new tool called *meta-linker*. In both cases the compilation model and the linking method that are used to produce the executable code are modified with respect to the basic model described in section 4.

5.2 Custom linking script

Compilation model. The current practice when using common compilation toolchains is to have all applications in the same computer node compiled as a single Ada program. Spatial isolation can be achieved if all applications are programmed according to a set of rules that clearly mark the tasks and data belonging to each application, e.g. using new pragmas or formal annotations. An ASIS [13] tool can then be used to check the source code and detect possible problems at a system-wide level and to generate a custom script that is used by the linker to produce an appropriate memory map (figure 4).

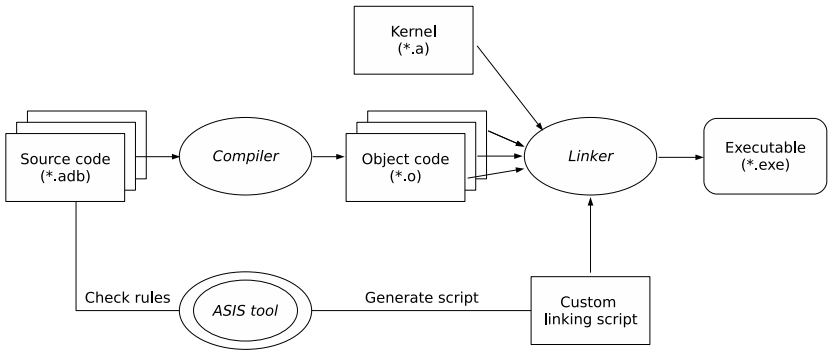

Fig. 4. Approach 1: Custom linking-script.

This approach requires a precise set of Ada rules for partitioning to be defined. Some rules are straightforward, e.g. "tasks belonging to different applications may not be declared in the same package", but some others are more complex, e.g. those on data types transmitted to other applications in order to avoid cross-partition pointers. Overall, a set of rules similar to the Ada Distributed Systems Annex (DSA) [3, App. E] can be defined, with the difference that the run-time system can be shared among all the partitions in the same computer node.

Protected objects (marked with a specific pragma) can be used for inter-partition communication (note again footnote 1). Such objects are located in a specific shared memory region, independent of those allocated to partitions. However, when the *proxy-model* implementation of protected objects is used (as in e.g. GNAT), a task can execute some entry code on behalf of some other task

[14]. This means that the proxy task may need to write some results in a stack belonging to another partition. One possible solution is to forbid `out` parameters in protected entries that are used for inter-partition communication. In this case, the entry is used only for signalling the arrival of an inter-partition message, and a protected procedure is then called to read the data.

Linking method. The linker binds each symbol (subprogram or global data) to a specific memory address [15]. This first approach relies on using an appropriate linking method for partitioning code and data into disjoint memory areas, in order to be able to take advantage of hardware memory protection. This approach also requires the kernel to be slightly modified so that it creates the stack of each thread in the global data area allocated to its partition.

The simplest way to implement this approach is to make an ASIS tool that checks the programming rules and generates a custom *linking script* for the system. The script specifies the location of each piece of data and each memory stack according to the partition it belongs to. The linker uses this custom script to generate an executable image with code and data allocated to the specified areas and symbol resolution (see figure 4).

An important advantage of this strategy is that existing response time analysis techniques can still be used. However, a new set of complex programming rules needs to be defined in order to provide partitioning among applications, and some of them may not be amenable to efficient static checking. In addition, the tool must support all the programming languages used in the system, which may be infeasible in some cases.

5.3 Meta-linker

Compilation model. The second strategy for memory isolation is based on compiling each partition as a separate Ada program, with all its tasks and global data belonging to that partition. Task priorities are global, i.e. the scheduler does not have any notion of partitions. On the other hand, no global variables can be shared among partitions. Hence, a new kernel service for inter-partition communication, similar to a message queue, has to be implemented. This service can be specially crafted to be very efficient in CPU time and memory space. Only one-way communication is needed, so blocking time can be minimized with respect to intra-partition synchronization primitives. The main requirement is that inter-partition communications must be predictable so that response time analysis can still be performed.

The problem with this approach is how to perform application-wide analysis with applications running in multiple partitions. Since there are no global shared data, the Ada Distributed Systems Annex can be used as a basis. Notice that the DSA also supports partition-wide strong typing enforcing by the compiler. The DSA is designed so that each partition has a separate run-time system. However, in this case the run-time can be shared among all partitions in order to reduce memory footprint. Distribution transparency is achieved as any partition can

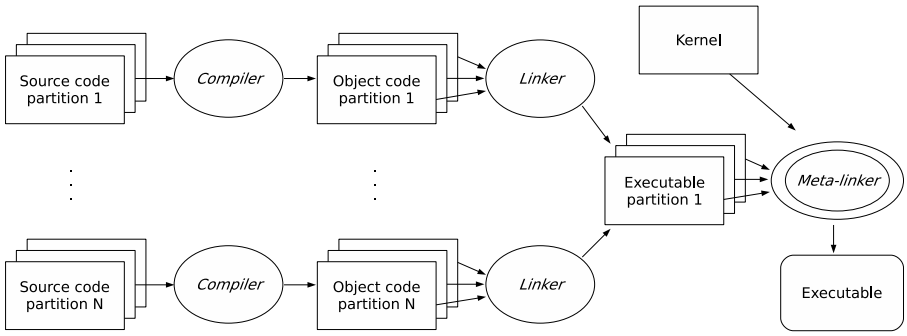

Fig. 5. Approach 2: Meta-linker.

be moved to another node without source code modification. Therefore Ada provides all the needed support, and there is no need for language extensions like new pragmas as in the previous approach.

Linking method. Under this approach each partition is first linked separately, but relocation information is retained in a linkable output format (e.g. ELF offers this possibility). Then a new tool called *meta-linker* finally sets the memory area of each partition, the kernel, and all shared libraries (including the Standard Ada Libraries) into a single executable. The data of each partition are bound to a separate location by the meta-linker so hardware memory protection can be used, taking into account the size and alignment requirements. The meta-linker also creates some data structures describing the layout of the partition. This information is needed by the kernel to adjust the fence registers in each context switch.

The meta-linker can be seen as an alternative to the address translation performed by an MMU, although in this case the address translation is performed statically within the compilation chain. It can be simple enough to be a qualified tool [6, §12.2], and therefore there is no need to certify again all the partitions if a change in the size of one of them results in modifying their base addresses.

It is often required that a partition can be independently modified without a need for re-linking and re-certifying other partitions. This can be done in two ways. The first one is that the linker resolves all symbols with an arbitrary base address, retaining all the relocation information for each symbol in the object code. The meta-linker adjusts the base address of all symbols for each partition at the time of building the final executable. The alternative is to generate Position Independent Code (PIC), so that the meta-linker only needs to adjust some symbols and specific pointers to global data. The problem with the first solution is that it makes the meta-linker more complex, and it also may require some changes in the compiler and linker, this making qualification more difficult. On the other hand, compiling as PIC often leads to larger and slower code, which may be a problem when computing resources are scarce.

5.4 Prototyping

In order to evaluate performance penalties, a meta-linker prototype and a new version of GNAT/ORK for LEON (a variant of GNAT[4] which uses an evolved version of the ORK kernel [16]), which can generate PIC, have been built.

The first problem that has been investigated is a potential increase in footprint. Preliminary measurements have been taken in order to evaluate the differences between PIC and non-PIC executables. Compiling both synthetic benchmarks and real code used in space projects, the increase in the number of instructions can be considered tolerable for this type of embedded systems. The size increase of the executable code (`.text` section) has been found to be about 7–15%, and the penalty in the total memory usage is only 1–4%, including also the data and stack segments, as shown in table 1.

Table 1. Footprint increase of Position Independent Code (PIC)

	Executable code		Global data		Stacks	Total	
	non-PIC	PIC	non-PIC	PIC		non-PIC	PIC
Benchmark 1	59 KB	67 KB	383 KB	384 KB	220 KB	661 KB	670 KB
Benchmark 2	104 KB	118 KB	385 KB	387 KB	420 KB	908 KB	925 KB
Application 1	439 KB	478 KB	422 KB	429 KB	320 KB	1181 KB	1227 KB
Application 2	1060 KB	1134 KB	599 KB	610 KB	320 KB	1979 KB	2064 KB

Position Independent Code (PIC) also has an execution time penalty when calling to a function in a shared library or referring to a global variable. As the linker cannot know where the code will be loaded, non-static routines must be called via a Procedure Linkage Table (PLT) and two actual jumps are performed instead of just one as usual. In order to measure the negative impact of on performance, a set of composite benchmarks and high-level algorithms from the Performance Issue Working Group (PIWG) test suite were used. Table 2 shows the results which for the Dhrystone and Whetstone benchmarks, as well as three complex algorithms. The significant differences are just in the Dhrystone and Whetstone benchmarks, as the differences in the high level algorithm tests and other PIWG tests are negligible or even favour PIC.

The most significant difference is in the Dhrystone benchmarks with full optimization where the performance penalty of PIC is about 38%. The penalty is about 21% for this test with no optimization. However, the Dhrystone benchmark consists of composite calls to integer routines with a very short execution time. Conversely, the Whetstone benchmark routines perform floating point calculations with considerably longer execution times. In this cases, the penalty ranges from 0.5% to 4.5%. Of course, a program calling mostly short routines will pay a comparatively higher penalty due to extra jumps.

The real situation is likely to be closer to the high level algorithms, where the maximum penalty is about 12%, and the minimum one is negligible. Therefore,

[4] `http://www.adacore.com/`

Table 2. Comparison in execution time.

Description	No optimization		Full optimization	
	non-PIC	PIC	non-PIC	PIC
Dhrystone	91.50 μs	111.52 μs	30.56 μs	41.66 μs
Whetstone manufacturers math routines	228.00 ms	233.50 ms	128.62 ms	131.88 ms
Whetstone with built-in math routines	207.76 ms	208.76 ms	66.68 ms	69.56 ms
NASA Orbit determination	586.00 ms	635.50 ms	281.50 ms	316.50 ms
JIAWG Kalman benchmark	185.76 ms	186.00 ms	20.28 ms	20.18 ms
Tracker centroid algorithm	5.64 ms	5.58 ms	2.08 ms	2.08 ms

it can be said that the penalty of using PIC is acceptable both in footprint and performance for typical real situations. It must be noticed that using an MMU approach for spatial isolation also pays a significant performance penalty due to heavier context switches.

In summary, it can be said the best way to provide spatial isolation based on fence registers as the only hardware support is the second proposed strategy, i.e. writing separate source code for each application and compiling and linking each partition separately, keeping the relocation information. A qualified meta-linker is then used to examine the sizes of the kernel, the shared libraries, and the partitions, adjust the base address of the whole application, and generate a single binary image. Finally, the real-time kernel adjusts the fence registers and processor mode at run time in order to provide the required strong hardware memory protection between partitions.

This is an elegant and powerful solution as it enables distributed applications (e.g. using a specialized DSA implementation) to be written in any programming language, including Ravenscar Ada and SPARK for high-criticality applications, or full Ada and C for low-criticality ones. It enables the performance and predictability problems of an MMU to be avoided, and allows individual partitions to be modified without having to certify again the whole system.

No modifications are required to current compilers, assemblers, or linkers, and the meta-linker is designed to be simple enough to be qualified for the development of high-integrity software. Furthermore, no extensions are required to the Ravenscar profile for enabling spatial isolation using the meta-linker approach. The few and localized additions to the kernel are not expected to hinder certification, being less complex or at least comparable to the software implementation support required by an MMU.

6 Related work

The implications of the MMU in Integrated Memory Avionics have also been studied by Audsley and Bennet [12]. Using SPARK for mixed criticality high-integrity systems was proposed by Amey and others [9].

The performance penalties with respect to Position Independent Code have been analysed by several authors. However, measurements comparing PIC and

non-PIC executables for embedded systems are not easy to find. One example of measurements for general purpose C++ applications is presented by Hamilton [17].

Other industrial domains could take advantage of the proposed techniques for achieving spatial isolation. For example, Autosar [18] is an automotive standard with a similar objective as Integrated Modular Avionics. The target CPUs used in those systems do not usually have an MMU, and therefore the standard does not consider spatial isolation. However, some of the techniques proposed in this paper can be a solution to provide memory protection on such systems.

7 Conclusions

Spatial isolation is needed to comply with the requirements of the the next-generation systems in the aerospace domain. A Memory Management Unit is commonly used for this purpose in general purpose operating systems, but performance and predictability problems appear when using MMUs in hard real-time embedded systems. Indeed, processors currently used in the European space industry an other embedded application domains, only include basic memory protection mechanisms, such as fence registers.

Several techniques have been explored in order to find a memory isolation scheme that can be used in this type of systems, most of them taking advantage of the unique characteristics of the Ada language. The recommended approach for systems composed only of high-integrity code is to use a safe subset of the language, such as SPARK, which also enables the absence of errors to be statically proved under appropriate conditions.

For systems composed of high- and low-criticality applications, a novel and powerful solution, involving a separate compilation of each partition, and a qualified meta-linker to generate the final executable, has been proposed. This flexible approach provides the same features as traditional techniques like strong memory partitioning, independent certification of partitions and maintenance, but it requires less hardware functionality and adds less overhead as specific processing is done statically at build time. Finally, additional Ada run-time checks can be used to detect programming errors inside each partition. A special-purpose implementation of the Ada Distributed Systems Annex can be used to enable static program-wide analysis of applications spanning multiple partitions, a characteristic which is often required for the certification of high-integrity systems.

A meta-linker prototype has been implemented as a proof of concept of the whole approach. The tool is simple enough to be qualified to a high-integrity level, and experimental performance and footprint metrics show that there is not a substantial penalty if the partitions are compiled as Position Independent Code. No modifications are required to the compiler, assembler or linker.

Future work includes specific compiler modifications to improve the generation of position independent code for embedded platforms, and research about how to reduce the impact of processor mode changes in space processors.

References

1. ECSS: ECSS-Q-80B Space Product Assurance — Software Product Assurance. (2003) Available from ESA.
2. Rushby, J.: Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center (June 1999) Also to be issued by the FAA.
3. ISO/IEC: Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652:1995/Amd 1. (2007) Published by Springer-Verlag, ISBN 978-3-540-69335-2.
4. Pulido, J.A., Urueña, S., Zamorano, J., de la Puente, J.A.: Handling temporal faults in Ada 2005. In Abdennadher, N., Kordon, F., eds.: Reliable Software Technologies — Ada-Europe 2007. Number 4498 in LNCS, Springer-Verlag (2007) 15–28
5. Zamorano, J., de la Puente, J.A., Hugues, J., Vardanega, T.: Run-time mechanisms for property preservation in high-integrity real-time systems. In: OSPERT 2007 — Workshop on Operating System Platforms for Embedded Real-Time Applications, Pisa. Italy (July 2007)
6. RTC: RTCA SC167/DO18B — Software Considerations in Airborne Systems and Equipment Certification. (1992) Also available as EUROCAE document ED-12B.
7. ARINC: Avionics Application Software Standard Interface — ARINC Specification 653-1. (October 2003)
8. Gaisler Research: LEON2 Processor User's Manual. (2005)
9. Amey, P., Chapman, R., White, N.: Smart certification of mixed criticality systems. In Vardanega, T., Wellings, A., eds.: Reliable Software Technologies — Ada-Europe 2005. Volume 3555 of LNCS., Springer-Verlag (2005) 144–155
10. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. ACM SIGOPS Operating Systems Review **27**(5) (December 1993) 203–216
11. Chang, Y.J., Lan, M.F.: Two new techniques integrated for energy-efficient TLB design. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **15**(1) (January 2007) 13–23
12. Bennett, M.D., Audsley, N.C.: Predictable and efficient virtual addressing for safety-critical real-time systems. In: Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS 2001), IEEE Computer Society Press (June 2001) 183–190
13. ISO: Ada Semantic Interface Specification (ASIS). ISO/IEC- 15291:1999. (1999)
14. Giering, E.W., Baker, T.P.: Implementing Ada protected objects—interface issues and optimization. In: TRI-Ada '95: Proceedings of the conference on TRI-Ada '95, New York, NY, USA, ACM Press (1995) 134–143
15. Levine, J.R.: Linkers and Loaders. Morgan Kaufmann (January 2000)
16. Urueña, S., Pulido, J.A., Redondo, J., Zamorano, J.: Implementing the new Ada 2005 real-time features on a bare board kernel. Ada Letters **XXVII**(2) (August 2007) 61–66 Proceedings of the 13th International Real-Time Ada Workshop (IRTAW 2007).
17. Hamilton, G., Nelson, M.N.: High performance dynamic linking through caching. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA (1993)
18. Heinecke, H., Schnelle, K.P., Fennel, H., Bortolazzi, J., Lundh, L., Leflour, J., Maté, J.L., Nishikawa, K., Scharnhorst, T.: AUTomotive Open System Architecture — an industry-wide initiative to manage the complexity of emerging Automotive E/E-Architectures. In: Convergence 2004. (2004)