Memory Isolation in Many-Core Embedded Systems[⋆]

Juan Zamorano and Juan A. de la Puente

Universidad Politécnica de Madrid (UPM),

Abstract. The current approach to developing mixed-criticality systems is by partitioning the hardware resources (processors, memory and I/O devices) among the different applications. Partitions are isolated from each other both in the temporal and the spatial domain, so that low-criticality applications cannot compromise other applications with a higher level of criticality in case of misbehaviour. New architectures based on many-core processors open the way to highly parallel systems in which each partition can be allocated to a set of dedicated processor cores, thus simplifying partition scheduling and temporal separation. Moreover, spatial isolation can also benefit from many-core architectures, by using simpler hardware mechanisms to protect the address spaces of different applications. This paper describes an architecture for many-core embedded partitioned systems, together with some implementation advice for spatial isolation.

Keywords: Embedded systems, mixed-criticality systems, integrated modular avionics, many-core real-time systems

1 Introduction

Mixed-criticality systems are composed of several subsystems with different levels of criticality as defined in some specific standard, such as IEC 6158, EN 50128, DO178C, and ISO 26262. Components with high criticality levels usually require certification by some independent organization, which often implies a complex and costly verification and validation (V&V) process. Since certification is generally carried out at system level, unless specific arrangements are made all the system components must be certified to the highest criticality level in the system. This is often unfeasible, as lower criticality subsystems may include COTS components that are not amenable to a strict V&V process, and almost always unpractical, as the cost of certification may be unacceptably high.

A common approach to overcoming this problem and keeping certification costs at a reasonable level is to provide *temporal and spatial separation* between components of different criticality levels, thus preventing a misbehaving

[⋆] This work has been partially funded by the Spanish Government, project HI-PARTES (TIN2011-28567-C03-01), and by the European Commission FP7 programme, project MultiPARTES (IST 287702).

low-criticality component from jeopardizing the temporal behaviour of high-criticality components or accessing their storage space. This enables independent certification of critical subsystems only.

Partitioned systems implement this concept by having a number of partitions running on a shared computer platform. Each partition hosts a different subsystem with a given criticality level. A *separation kernel* takes care of implementing and spatial separation by scheduling the execution of partitions in separate time frames and providing isolated memory spaces for them. A well-known example of partitioned architecture is defined in the ARINC 653 standard for Integrated Modular Avionics (IMA).

Virtualization provides a means to provide a number of *virtual machines* (VM) on a single hardware platform. Each virtual machine has a set of virtual resources that are mapped to the available physical resources. It is generally accepted that the best approach to virtualization in real-time embedded systems is based on the use of a *hypervisor* or virtual machine monitor [6] that divides the available physical resources into the different virtual machines. This technique can be used to implement partitioning, by making each virtual machine a separate partition, possibly with a different operating system depending on the criticality requirements of the subsystems, or applications, running on it (figure 1).

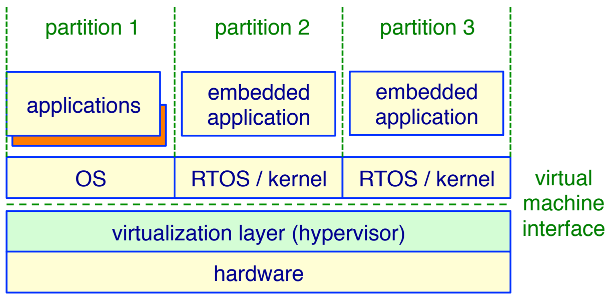

Fig. 1. Virtualization and partitions.

This approach is being used in the MultiPARTES[1] and HI-PARTES[2] projects to implement mixed-criticality embedded systems on multi-core platforms [7] based on the XtratuM,[3] open-source hypervisor [5, 3]. In this paper we analyse the implications of further extending partitioning to many-core platforms, and propose a new approach to spatial separation in this kind of systems. The approach is based on a previous development for mono-processor platforms [8]. Some complementary views on temporal separation in many-core systems were presented in a previous paper [9].

[1] `www.multipartes.eu`
[2] `www.dit.upm.es/str/hi-partes`
[3] `www.xtratum.org`

The rest of the paper is organized as follows: current techniques for implementing spatial separation in partitioned systems are reviewed in section 2. An alternative approach using limited hardware support is presented in section 3, and then applied to many-core systems. Implementation issues are discussed in section 4. Finally, some conclusions and future work plans are described in section 5.

2 Spatial Isolation in Partitioned Systems

Current memory isolation techniques for partitioned systems are commonly based on using Memory Management Units (MMU), implemented in hardware, to prevent software running in a partition from reading or writing into address space allocated to other partitions. MMUs can provide sophisticated memory management schemes as they are designed to support complex paging and virtual memory in general-purpose operating systems. They provide address translation mechanisms to map the logical memory space of an application to the regions of physical memory that are allocated to it. Some parts of the logical address spaces may actually reside in secondary storage, but this feature is not commonly used in embedded real-time systems because it leads to a high degree on unpredictability in the execution time of real-time tasks. Therefore, in this kind of systems the logical address spaces of partitions are always mapped to physical main memory areas, and MMUs are used for allocation, protection and address translation.

A key element for MMUs are translation look-ahead buffers (TLBs), a set of fast registers that are used as caches for address translation, in order to avoid significant access time penalties in memory operations. Nevertheless, despite their utility for this purpose, TLBs have some drawbacks. First of all, they have a high level of power consumption [2]. Moreover, the possibility of TLB misses hinders the predictability of the system and introduces some overhead due to address translation and TLB flushes [1]. TLB flushes may be needed in every partition context switch if TLB entries are not tagged with the partition identification. This approach also requires the allocation of part of the partition physical memory to the page tables needed for translation and protection.

The MMU-based space isolation approach is depicted in figure 2. When the processor starts a read or write operation on a logical address in the logical space of the currently running partition, the MMU translates it to a physical address in the main memory. The TLB tag memory is accessed to find the corresponding physical address, if there is a hit then the access is validated. Otherwise, if there is a miss the page tables must be traversed resulting in a great time penalty, and the corresponding TLB entry must be updated. Finally, if the access is within the partition private memory and has the required permissions, the physical address is accessed. If not, the hypervisor processes an invalid access exception and takes proper measures to isolate and handle the failure.

The physical memory region assigned to a partition must contain the partition page tables that are needed for address translation and memory protection.

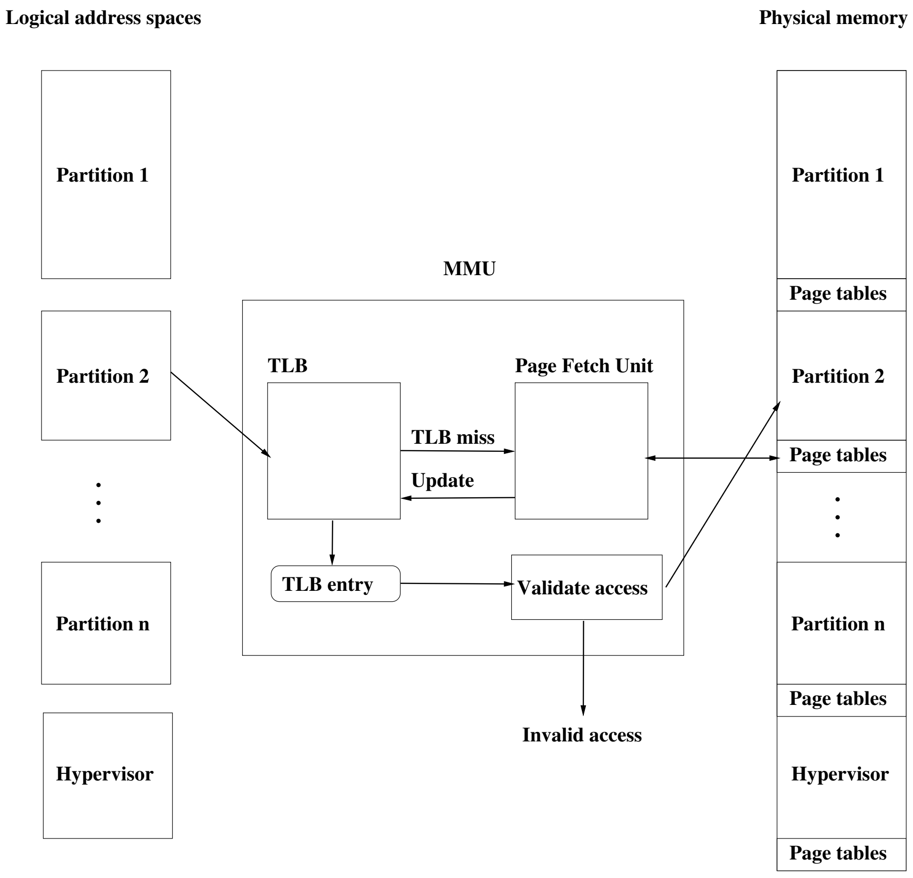

Fig. 2. Memory isolation with memory management unit.

The page tables have to reside in the physical memory of the partition because they have to be accessed by the page fetch unit to fill the TLB entries. Moreover, paging systems suffer internal fragmentation because the physical memory must be allocated in page size units. For these reasons, the actual amount of physical memory that has to be allocated to a partition is larger than the size of its logical address space, as some additional amount of the available physical memory is used to support this approach.

3 Protection-based spatial separation

3.1 Memory protection

As above said, embedded systems usually have real-time constraints, and for this reason the whole logical address space of each partition is allocated in physical memory. Therefore, page faults can not occur, and this source of indeterminism is removed from the system. This makes a crucial difference between embedded and general-purpose systems for which MMUs were primarily developed.

This arrangement enables other strategies for memory isolation, such as compiling each partition as a separate program that is linked separately for a predefined region of physical memory, as usual in embedded system development. A second linker then links all the executable files in the same logical address space into a single file that fits the available physical memory space [8].

The XtratuM hypervisor uses a configuration file where memory segments are specified, and virtual devices and physical peripherals are allocated to each partition. [4] Partitions are linked into predefined memory segments, and then a system image is built from partition and hypervisor binaries. The system image is loaded into physical memory according to the memory layout defined in the configuration file. Therefore, address relocation is done as part of the system development, and address translation is not needed at run-time.

It must be noticed that some kind of memory protection is still needed to prevent unauthorized access to the parts of the logical address space that are private to other partitions. However, a fully featured MMU is not needed, and the required memory protection mechanisms can be implemented with less hardware support. Using a simpler MPU (Memory Protection Unit) is enough to prevent partitions from accessing memory beyond their allocated physical area. The MPU validates that every access is within its assigned physical memory segment, and otherwise generates an invalid access exception that can be dealt with by the hypervisor as before. A classical way of implementing an MPU is using a pair of fence registers to store the limits of the physical memory region allocated to the partition. Each address is checked to be within the limits defined by the fence registers, and otherwise an exception is raised.

The MPU-based approach to spatial separation is depicted in figure 3. The logical addresses generated by the processor are now the same as physical addresses, as there is no address translation. The MPU checks that the address is

[4] See `www.xtratum.org` for more details.

within the limits of the memory segment of the partition. If the check fails, the hypervisor processes the invalid access exception and takes proper measures to isolate and handle the failure.

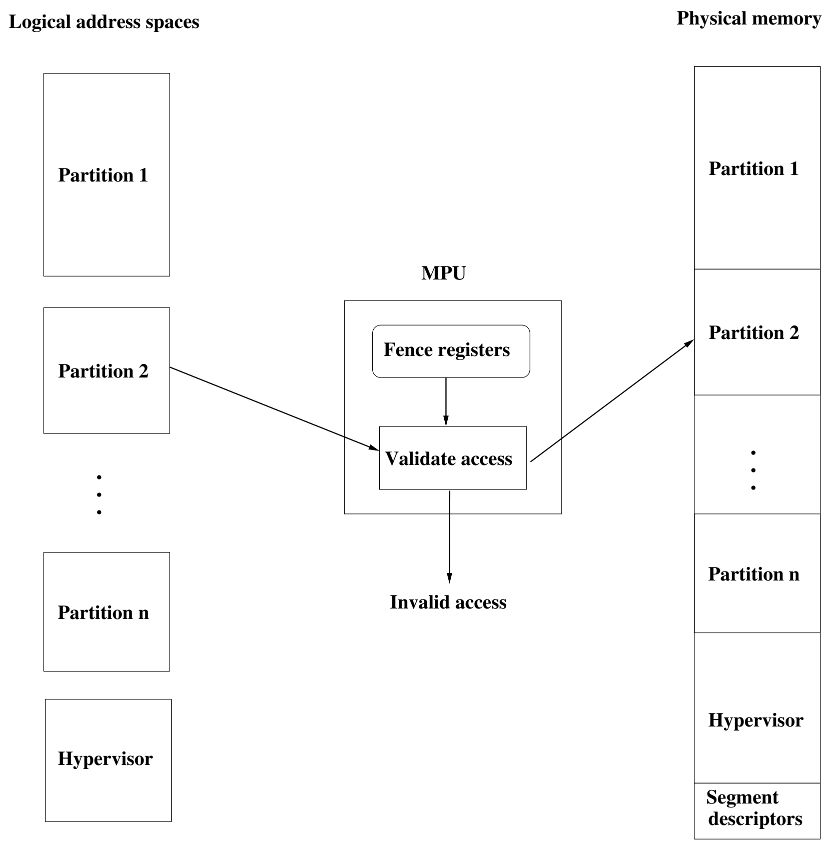

Fig. 3. Memory isolation with memory protection unit.

It should be noticed that the physical segments assigned to the partitions have the same size as the corresponding logical spaces. The segment descriptors that contain fence register values are stored in the hypervisor space, since it is in charge of loading the new values into the MPU at partition context switches. Only two values are needed to protect a segment, and therefore the amount of physical memory needed for segment descriptors is much less than in the MMU-based approach. Moreover, segment sizes need not be multiples of a predetermined page size, and thus there are no memory leaks due to internal fragmentation. As a result, the amount of physical memory that is needed to support spatial isolation is minimized.

3.2 Application to many-core systems

The advent of many-core processors can be expected to simplify processor scheduling, as in a scenario with more cores than partitions, processors will not have to be multiplexed among partitions. Each partition can be allocated to a set of dedicated processor cores, and the role of the hypervisor with respect to temporal separation can thus be greatly simplified. Figure 4 shows how a mixed-criticality system can be implemented as a partitioned system on top of a many-core platform.

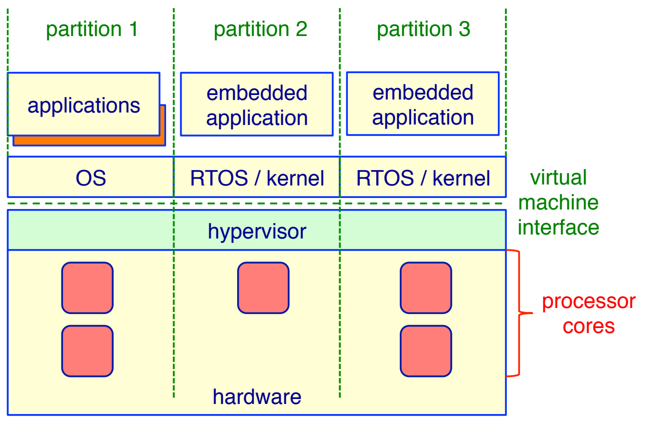

Fig. 4. Partitioned many-core systems.

Indeed, as many-core architectures evolve towards a growing number of processor cores, a fully parallel approach is getting more interesting for mixed-criticality systems. Having one or more physical cores dedicated to each partition simplifies scheduling and eases temporal separation [9].

This static assignment of processor cores to partitions also simplifies the hardware and the software needed to support spatial isolation. As only one partition and the hypervisor can execute in each processor core, only the private memory of the partition and the hypervisor memory need to be accessed by the core. Therefore, only two contexts are needed to retain translation (MMU) or protection (MPU) for the two address spaces. This simplifies the implementation of the hardware supporting spatial separation, as well as the design of the hypervisor that is in charge of assigning, reclaiming and reassigning the available contexts among the partitions. Contexts available in a core are statically assigned at boot time.

In the case of MPUs, this means that only a pair of fence registers are needed to protect a partition memory segment. It should be noticed that hypervisors run in supervisor mode, and thus can access the whole physical memory. Therefore only faulty partitions can attempt to access memory addresses out of their allocated memory. However, since a partition may have physical devices allocated to it, additional pairs of fence registers may be needed to detect faulty accesses in the I/O address space.

4 Implementation aspects

4.1 Many-cores on FPGA devices.

Nowadays, FPGA (Field-Programmable Gate Array) devices are widely used to implement SoC (System on a Chip). FPGA devices are built with a number of predefined logic blocks that can be programmed to implement required functionalities. Their interconnections can also be programmed as well as the I/O pads that enable accessing other devices in the FPGA board such as memories, peripherals, etc. As any other silicon chip, the number of available resources in the device is constantly growing and the implementation of many-core systems on FPGA devices can be foreseen in the near future.

Some libraries of IP cores, such as GRLIB[5] allow SoC and NoC to be synthesized on FPGA. This library includes processor cores, such as LEON3 processor and LEON4, which can be used to synthesize multiprocessor systems on an FPGA.

As experiment, we have synthesized LEON3 processors with and without an MMU in order to evaluate the cost of implementing the MMU. We have used two families of FPGA devices, namely the Actel ProASIC3 and the Xilinx Virtex-5. Both families have different toolchains that provide different information about the resource utilisation. Table 1 contains a summary of the FPGA resource usage of both designs for Actel ProASIC3 FPGA devices, and table 2 shows the results for Xilinx Virtex-5 FPGA devices.

	Core Cells	Block Rams	IO Cells
LEON3 without MMU	6710	4	58
LEON3 with MMU	8895	6	58
Increment	32%	50%	0

Table 1. Resource utilisation for Actel ProASIC3 FPGA devices.

	Registers	LUTs	Flip Flop	IO	Specific
LEON3 without MMU	857	1044	1168	198	8
LEON3 with MMU	1076	1213	1420	198	8
Increment	25%	16%	21%	0	0

Table 2. Resource utilisation for Xilinx Virtex-5 FPGA devices.

[5] `www.gaisler.com/index.php/products/ipcores`.

The results are not easy to interpret in terms of the overall increment in silicon usage that is required to implement a processor with MMU, since the different technologies give different numbers. Neither of the technologies has a penalty for I/O blocks, as both designs have the same I/O interface. However, it can be roughly deduced that the LEON3 MMU has an extra cost in logic blocks of about 30 % with the Actel technology, and about 20 %with the Xilinx technology. Unfortunately, there are not IP cores available for fence registers in GRLIB, and it is not possible to directly valuate how many extra logic blocks would be needed for a LEON3 with an MPU. Nevertheless, given that fence registers are much simpler than MMUs or processors, it can be assumed that the cost would be negligible.

We can then conclude that a LEON3 with an MMU has an extra cost of approximately 20% with respect to the same processor and an MPU. For example, FPGA devices that can contain 10 LEON3 with MMUs can be expected to accommodate about 12 LEON3 with MPUs.

4.2 MPUs in embedded processors.

In spite of the fact that MPUs enable simpler and more efficient spatial isolation techniques to be used, most embedded processors include full MMUs. The main reason for it is that this kind of processors are usually built as microcontrollers, i.e. general-purpose processors with integrated memories and peripherals. However, some families of embedded processors with MPUs that can be used to implement our approach can be found in the market.

For instance, the Cortex-M0+, Cortex-M3 and Cortex-M4 microcontrollers, based on different versions of the ARM architecture, support an optional MPU.[6] The Cortex MPU can protect up to eight memory segments with sizes of 2^n bytes where n ranges from 5 to 32. Every segment can be split into eight subsegments of the same size. Segment protection includes privilege level and access rules. Attempts to access an invalid address raises an invalid access exception.

Some Freescale microcontrollers, based on the 32-bit Power Architecture, such as the Qorivva MPC5643L,[7] include an MPU in addition to an MMU. The use of the MPU is recommended for safety-critical systems, such as those qualified to ASIL D level in ISO 26262 [4]. The MPU supports access control by using region descriptors that define memory segments and their associated access rights.

Older members of the LEON family of radiation hardened processors, such as LEON2, have two pairs of fence registers for memory access control.[8] This implementation of the MPU concept, though, is not complete, as read accesses out of a segment are allowed.

The Intel x86 family of processors[9] uses segmentation as the basic mechanism for memory protection. Starting from the i80386 processors, paging was added in

[6] `www.arm.com`
[7] `www.freescale.com`
[8] See `www.gaisler.com`.
[9] `www.intel.com`

order to mitigate segmentation problems in virtual memory systems. However, the paging unit can be disabled and thus pure segmentation can be used of memory protection.

5 Conclusions and future work

The trend towards using many-core processors in embedded systems, and the need to run applications with mixed criticality levels on the same computer platform naturally lead to the concept of partitioned many-core systems. The mid-term scenario is one where there are more processor cores than partitions, thus making unnecessary the use of partition scheduling methods such as the kind of static scheduling used in current partitioned architectures.

We have shown in the previous sections that address translation is not needed for spatial separation in partitioned embedded systems, as virtual memory is generally not used. Therefore, the complexity and penalties in execution time and power consumption incurred by using a conventional MMU are an unnecessary burden in this kind of systems.

An alternative approach is to use MPUs with fence registers to delimit the memory areas allocated to partitions and provide spatial separation between them. This technique was first proposed for monoprocessor systems, combined with link-time logical to physical address mapping for intra-partion code. The technique uses a dual-linker toolchain to generate the executable code of the partitions.

This approach can be extended to multi-core and many-core systems. In the latter case, the hypervisor can be greatly simplified by statically allocating partitions to one or more cores, so that each core runs only code from one partition and the hypervisor. Therefore, only two contexts are needed on a core, and fence registers have only to be loaded at boot time.

From the implementation point of view, we have found that the MPUs that are currently available in some embedded processors are overly complex, for they are intended to protect more memory segments than required in our approach.

As a future work we plan to develop an IP core with fence registers for GRLIB that can be used to build a simple MPU. In this way, it will be possible to synthetize SoC with LEON processors including the simple kind of MPUs that have been discussed above, in order to accurately evaluate the amount of FPGAs resources that are needed for this approach. The XtratuM hypervisor can then be modified using the proposed protection mechanism, so that the benefits of the approach can be further assessed.

References

1. 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). pp. 183–190. IEEE Computer Society Press (June 2001)

2. 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), 13–23 (January 2007)
3. Crespo, A., Ripoll, I., Masmano, M.: Partitioned embedded architecture based on hypervisor: The XtratuM approach. In: European Dependable Computing Conference — EDCC 2010. pp. 67 –72 (april 2010)
4. Freescale Semiconductor: Safety Application Guide for Qorivva MPC5643L (2012)
5. Masmano, M., Ripoll, I., Crespo, A.: An overview of the XtratuM nanokernel. In: OSPERT 2005 — Workshop on Operating System Platforms for Embedded Real-Time Applications. Palma de Mallorca (July 2005)
6. Rosenblum, M., Garfinkel, T.: Virtual machine monitors: current technology and future trends. Computer 38(5), 39 – 47 (may 2005)
7. Trujillo, S., Crespo, A., Alonso, A.: MultiPARTES: Multicore virtualization for mixed-criticality systems. In: Euromicro Conference onDigital System Design, DSD 2013. pp. 260–265 (2013)
8. Urueña, S., Pulido, J.A., López, J., Zamorano, J., de la Puente, J.A.: A new approach to memory partitioning in on-board spacecraft software. In: Kordon, F., Vardanega, T. (eds.) Reliable Software Technologies — Ada-Europe 2008. LNCS, vol. 5026, pp. 1–14. Springer (2008)
9. Zamorano, J., de la Puente, J.A.: On real-time partitioned multicore systems. In: 16th International Real-Time Ada Workshop (IRTAW 16) (April 2013)