Companion Chip: building a segregated hardware architecture

Thomas Pareaud [(1)], Alain Houelle, Nicolas Vaucher [(2)], Mathieu Albinet [(3)], Christophe Honvault [(4)]

[(1)] ASTRIUM Satellites SAS, 31 rue des Cosmonautes, 31042 Toulouse Cedex 4 – France
thomas.pareaud@astrium.eads.net
Phone: +33 (0)5 6219 8879
Fax: +33 (0)5 6219 7158

[(2)] Advanced Electronic Design, 3 rue de l'éperon, 77000 Melun – France
houelle@a-e-d.com, vaucher@a-e-d.com
Phone: +33 (0)1 64521696

[(3)] Centre National d'Etudes Spatiales, 18 av Edouard Belin 31401 TOULOUSE Cedex
mathieu.albinet@cnes.fr
Phone: +33 (0)5 61273248

[(4)] At ASTRIUM during the study, now at ESTEC
ESTEC, Keplerlaan 1 Postbus 299, 2200 AG Noordwijk – The Netherlands
Christophe.Honvault@esa.int
Phone: +31 (0)71 565 8181

ABSTRACT

Partitioning is a more and more mature concept in Space industry. It aims at assuring that some error propagation modes are not possible. This paper gives an overview of an analysis conducted in the frame of a research and technology study performed in 2010/2011. The "Java Companion Chip" study addresses an interesting approach to partitioning using hardware concepts: a SoC architecture integrates a master processor, a companion chip and additional hardware functions aiming at enforcing the time and space segregation between the master processor and the slave one.
This paper discusses the benefits and the main challenges of the proposed approach. In addition, it presents an application of these concepts to a case study: a Leon/Java processor architecture able to concurrently execute native and Java applications.

1. INTRODUCTION

CNES initiated the "Java Companion Chip" study in 2010 with the objectives of analysing and prototyping a multi-processor System on Chip (SoC) with hierarchical segregation of these processors.
Time and Space Partitioning (TSP) [1] aims at avoiding error propagation from one critical partition to a more critical one. It can be used in two main contexts: security [2] (only malicious fault are considered), and dependability [3] (all types of fault are potentially considered).
The Figure 1 reminds the partitioning principles [4]. The system is considered as a set of resources. In this context, a partition is a collection of subsets of those resources such that a resource belongs to exactly one partition. In other word, partitioning a system consists in exporting its resources (memory, binary, I/O, CPU time slot) into partitions such that a resource can not be owned by two partitions. An active resource (i.e. that can initiate a processing) is considered as a subject. In the picture, subjects are noted Sx and non-subject resources are noted Rx. The data flow is represented by the arrows that must involve at least one subject. This data flow is defined in the set of rules that composes the Partitioned Information Flow Policy (PIFP).

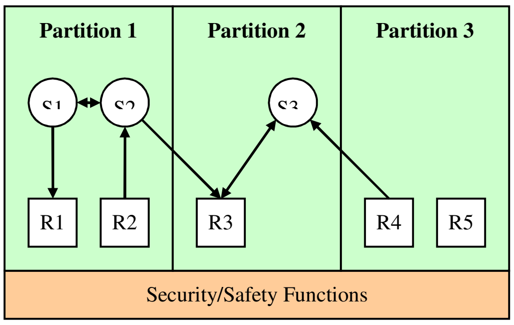

Figure 1: Partitioning principles

Two main manners are investigated to implement such a partitioned system:

- Virtualisation by hypervisors: the resources of the system are simulated by a hypervisor to create a virtual machine. This solution can be totally transparent for the content of the partition that runs as if it is on top of the real hardware. In this context, a virtual machine implements the partition concept.
- Separation kernels and micro-kernels: introduced by Rushby in [1], the partitioning is offered by an extended operating system that

provides stronger isolation between processes. In this context, a process implements the partition concept.

A direct consequence of partitioning is a lack of reactivity of the software to hardware I/O (or other inter-partition software events) that is the consequence of statically defined scheduling policy, necessary to enforce the time partitioning property. In addition, the software dynamic design can become much more complex due to the scheduling constraints with an important loose of efficiency (more frequent context switches, memory copies, etc).

In the same time, partitioning is becoming more and more popular and European Space community is preparing his new generation of processors. Efforts are being spent on studying multi-core / multi-processors solutions as in the “Next Generation Multipurpose micro-Processor” and “System Impact of Distributed Multicore Systems” ESA studies. Multi-cores solution seems an opportunity to increase parallelism and overall performance of the software. Meanwhile, it is raising many problems, especially regarding the determinism of the software execution with an improbable worst case execution time estimation, far from the mean execution time. This phenomenon mainly results from the introduction of hierarchical cache memory, and memory bus sharing among the different cores.

The work done in this study is in the between of these two current trends: it aims at providing hardware segregation between two processors, a master and a slave also called companion. Each processor executes its sets of functions having a different level of criticality, with a minimal loss of efficiency thanks to a multi-processor architecture allowing real software parallelism (see Figure 2). A case study is being developed on top of this architecture implementing a native Leon application (OBSW) with Java applications (OBCP) support. This case study is to be described in this paper.

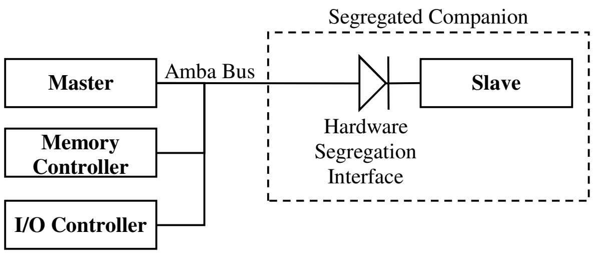

Figure 2: Segregated dual-core architecture

The paper is organised as follow. Next section details the context in which this study is performed and explicits the benefits and drawbacks of the proposed approach. The section 3 describes the SoC architecture and presents the hardware solution used to enforce the segregation between the two processors. The section 4 presents software considerations: how the software should rely on the hardware to improve its determinism when communicating with the companion, and, what is the expected impact on task’s worst case execution time (WCET) in order to being able to conduct schedulability analysis. The section 5 presents the use case of the study: a Java On-Board Control Procedure (OBCP) interpreter. The section 6 concludes the work performed in this study.

2. TECHNICAL CHALLENGES

The design of this architecture faces several technical challenges. The major ones are listed below.

Error propagation

Since the most critical software runs on the master processor, the architecture shall avoid error propagation from the slave processor to the master processor. One typical example regards data/code integrity. The data/code of the master software can be corrupted by the slave software. As a consequence, if an error impacts the slave software, it could propagate to the master software using memory access.

Performance optimisation

The architecture should benefit from the parallelism that is offered by the dual-core architecture. As a consequence, the segregation interface shall maximize the use of the shared bus.

Determinism

In a mono-master architecture, the processor does not compete for the Amba bus. Arbitration is obvious, and the time to access the bus is deterministic. In a multi-master architecture, some processors compete for the Amba bus. As in any problem of resource sharing, a policy must be defined to enforce the determinism of these accesses.

In conclusion, there is an important need for strong segregation that shall be enforced using hardware mechanisms. Design choices shall be guided by both error propagation, determinism and performance concerns.

3. ENFORCING SEGREGATION

The most efficient way to ensure the segregation and the determinism of both processors is to duplicate all resources and especially the memory. Nevertheless, efficient communication can not be done without a minimum of resource sharing (a bus for instance). The counter part of sharing resources is that reliability, security and determinism of the master processor could be impacted by the slave one.

In order to cope with this issue, a hardware interface (or wrapper) has been introduced for the slave processor. This interface aims at enforcing segregation between the slave and the master such that an error can not propagate from slave to master. The two considered propagation modes are:

- Time propagation mode: loss of the determinism of the master software leading to deadline miss. When the slave monopolises the bus, master software execution is slowed down such that, it may not be able to meet its timing requirements.
- Space propagation mode: Corruption of the Data/Code of the master software using direct memory access. Without any control, the slave can corrupt the code or data of the master.

Space propagation mode can be covered using a Memory Protection Unit (MPU). Its role is to control the address space to which the slave processor can access to. This MPU can be configured by the master processor in order to cope with a specific mission needs (start address, length).

Covering time propagation mode is equivalent to enforcing master software determinism independently of the software running on top of the slave processor. Determinism is always a problem in real-time applications. It means both:

- predictable (the time spent in processing a programme can be determined offline);
- and reproducible (from a same initial state, a programme will take the same time to be processed).

These two properties are very complex to implement. Most of the time, a simplification is done by only considering a worst case execution time. The main goal is to keep the worst case as close as possible to the mean. From the reproducibility point of view, a very representative problem is the impact of a slave memory access considering programmes with various execution times. Whereas the impact of a burst access from the slave as a negligible impact on the execution time of a large programme, it is not the case for a small one. From the predictability point of view, the approach should provide a method to determine offline the (worst case) impact on the execution time.

The selected approach consists in defining a memory access policy such that it has a deterministic impact on the master software execution time. To reach this goal, the hardware segregation interface implements two mechanisms:

- limitation of the burst size: a burst introduces a latency from the master point of view. This latency can be controlled by limiting the maximum slave burst size to a low value
- bus bandwidth quotas: Limiting the bus bandwidth that is available to the slave contributes to ensure an worst case performance of the master.

Such mechanism is implemented using two internal registers and one configuration register. Theses registers can have a value between 0 and MAX_WINDOW. The first internal register is an Amba bus cycle counter that wraps around every MAX_WINDOW cycles. It represents a time windows of MAX_WINDOW Amba bus cycles. The second internal register counts the number of Amba bus cycles used by the slave in less than MAX_WINDOW bus cycles. This register is reset every time the first internal register wraps around. The configuration register defines a threshold defining the maximal number of used cycles after which the slave can no more start transaction (burst) on the Amba bus. This mechanism thus defines a quota on a non-slicing time window.

Let take an example to illustrate such a mechanism with a MAX_WINDOW of 1024. The configuration register is set to 10% (102 cycles). The Amba bus cycle counter's current value is 523. The slave already used 101 bus cycles in the current time window and wants to access the Amba bus for a burst access of 8 words that takes in practice no more than 16 bus cycles. Then, the interface accepts to proceed the access since the current number of used cycles is below the configured maximum number of cycles (101<102). The access is performed and after this transaction, the updated number of used cycle becomes, in worst case, 117 cycles (101+16).

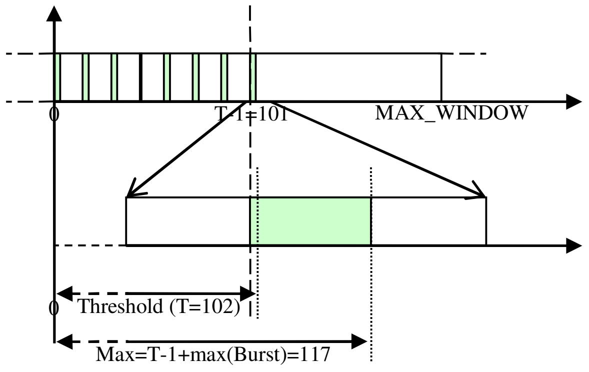

Figure 3: Quota policy implementation

In conclusion, the impact of the slave on the master is in worst case:

- An Amba bus access latency corresponding to the maximum number of bus cycles that is necessary to perform the greatest burst transfer (should be lower or equal to master cache line size).
- A slave quota on a non-slicing time window equals to:

$$(1)\quad MAX\_SLAVE = T - 1 + Max(Burst) \leq MAX\_CYCLE$$

$$(2)\quad Quota = \frac{MAX\_SLAVE}{MAX\_WINDOW}$$

4. SOFTWARE IMPACTS

4.1 Software design and determinism

In order to enforce the determinism of the software running on the Master processor, interactions between Master and Slave are subject to a limitation: the slave shall not introduce asynchrony on the master.
The simplest way to proceed is certainly not to use asynchronous interrupt handling mechanisms (that could lead to flooding behaviours) but to implement interrupt polling mechanisms. Then, the master software periodically polls the pending interrupt in order to process these interrupts synchronously.
Master to slave interaction is not subject to this limitation. A slave can use asynchronous mechanisms depending on its requirements.
This design choice thus impacts the way Master/Slave communications are implemented from the Master side. Nevertheless, in flight software, communication mechanisms are already implemented such a way. This solution is thus compatible with standard requirements and practices.

4.2 Impact on WCETs

The impact of the segregation interface on the master task's WCET depends on the value of three parameters:

- The MAX_CYCLES value: this value must be maintained sufficiently low to have a time window of the same order of acceptable task latencies. Nevertheless, its value should be kept far greater than the worst case of the greatest burst access of the slave.
- The greatest burst access: it has a direct consequence on the $WCET_{Burst}$ value. This value should be kept relatively small (lower or equal to the greatest master burst access). In practice, it is a good solution to use the same size than a cache line (e.g. 8 words).
- The threshold value: the threshold has a direct consequence on the available bandwidth for the master (see section 3).

From the Amba bus point of view, the less favorable case is obtained when the master task spent its time in accessing bus (which occurs for instance when writing in memory with the write-through cache policy of the Leon). The worst case of the real WCET of such a task is obtained when at the beginning and the end of the task, the bus is alternatively accessed by the master and the slave. An upper bound then is:

$$(3)\quad WCET_{real} \le \frac{WCET_{master}}{1-Quota} + 2\times MAX\_SLAVE$$

With:

$$(4)\quad \begin{aligned} &MAX\_SLAVE < MAX\_WINDOW << WCET_{master} \\ &\Rightarrow 2\times MAX\_SLAVE = o\left(\frac{WCET_{master}}{1-Quota}\right) \end{aligned}$$

We obtain the following acceptable simplification:

$$(5)\quad WCET_{real} \le \frac{WCET_{master}}{1-Quota}$$

5. CASE STUDY: JAVA OBCP

5.1 Motivations

On-Board Control Procedures is one technique among others to operate a satellite. It consists in writing a sequence of operations in a dedicate language which allow interacting with equipment and core functions of the satellite (Mission management, AOCS, etc). Such operating technique is really powerful and offers opportunity to enhance flexibility and operability in flight.
Management actions performed on OBCP in flight consist in loading new OBCP from ground, starting, stopping, suspending and resuming OBCP execution and removing OBCP. Indeed, OBCP can be developed after the launch of the spacecraft and dynamically loaded. Most OBCP are not subject to the same validation rigor than On-Board Software. The OBCP are managed and executed on-board by a dedicated engine (interpreter) that must ensure that errors cannot propagate from OBCP to Flight Software as well as between OBCP. Two solutions can be used to this aim:

- Use of a safe architecture: a safe architecture avoids the propagation of errors from OBCP engine to Flight Software. Such an architecture can mix hardware and software techniques. This is the case of the proposed segregated dual-core architecture proposed in this paper.
- Use of a safe language: some programming languages are safer than others in the sense that a more systematic verification approach is applied to avoid unhandled errors. This is the case of Java and it has been studied for a long time in AeroVM [5]

Java OBCP engine is thus a good case study to demonstrate the feasibility and opportunity of the dual-core segregated architecture proposed in this paper.

5.2 Selected Architecture

The SoC implementation integrates a Sparc/Leon3 master processor and the AED Java processor named JAP as the companion/slave processor.
The JAP processor has been designed to execute native and Java code. It has two instruction sets: native and Java. The native instruction set works in register mode (32 registers), Java instruction set works in stack mode.
The architecture of the JAP processor has been optimized to switch from one set to the other very quickly. JAP architecture characteristics:

- 32 bits processor

- Harvard architecture
- 8 interrupt line
- 4 pipeline stages
- Full IEEE-754 FPU

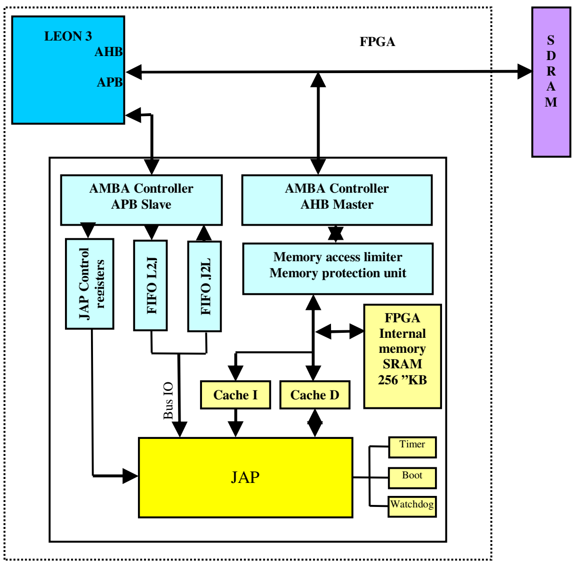

Figure 4: Overall case study architecture

The LEON/JAP architecture is configured in a FPGA Xilinx Virtex4. The LEON/JAP communications are ensured by two FIFOs. The FIFOs are connected to a slave APB AMBA controller. When the Leon write a data in the Leon to Jap (L2J) FIFO an interrupt is transmitted to the JAP. In order to not alter the FSW behaviour, the JAP to LEON (J2L) FIFO doesn't generate interrupt. The process which handles the JAP/LEON communications has to scan the state of the FIFO to know if data are available.
The slave AMBA controller also gives access to some state and control registers of the JAP processor, such as activation/deactivation of the JAP memory cache, reset of the JAP processor, memory access limiter, etc.
To minimize the accesses to the shared memory, the JAP processor has two caches: one for instruction (4K bytes) and one for data (4K bytes).
The SRAM of the FPGA is also used to store recurrent data. These data mainly includes the operating system which is often executed.
In order to secure the FSW, the memory protection unit checks if each shared memory access of the JAP processor is in the allowed area.
Finally, the "memory access limiter" limits the number of JAP accesses to the shared memory in order to not slow down the execution of the FSW.

6. CONCLUSION

This paper presented a segregated master/slave dual-core approach to cope with partitioning and processing power concerns. It explains how error propagation is avoided thanks to a hardware wrapper of the slave processor. By the end, it presents a case study that deals with Java On-Board Control Procedures interpretation where the OBCP engine is implemented on top of a Java processor. On going work is to validate such architecture and the predictions made on the performances. This validation shall be performed on functional aspects (communication mechanisms between Master and Slave), but also on non-functional aspects (determinism, error propagation) in order to validate predictions on execution times but also segregation property.

REFERENCES

1. J. Rushby, *Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance*. NASA Langley Research Center, 1999.
2. M. Leroy, "Invitation to Tender – Securely Partitioning Spacecraft Computing Resources," ESA.
3. ESA, "Integrated Modular Avionic for Space (AO/1-6295/09/NL/LvH)," 2010.
4. "U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, Version 1.03," 29-Juin-2007.
5. F. D. Bruin, F. Deladerrière, et F. Siebert, "A standard Java virtual machine for real-time embedded systems," *The Data Systems in Aerospace Conference (DASIA), Prague, Czech Republic*, 2003.