Shared Memory Protection for Spatial Separation in Multicore Architectures

Anton Hattendorf
fortiss GmbH
München, Germany
hattendorf@fortiss.org

Andreas Raabe
fortiss GmbH
München, Germany
raabe@fortiss.org

Alois Knoll
Institut für Informatik VI
Technische Universität München

***Abstract*—The introduction of multicore architectures in embedded systems allows system integrators to locate multiple applications on the same chip. In the context of certification separation of these applications is mandatory. Most current multicore systems have a low core count and programmers have a need for easily utilizable platforms. Therefore, most of the current multicore systems use shared memory architectures based on bus communication. In this paper we discuss several possible architectures for shared memory protection using local and shared MPUs and MMUs for architectures of this type. This analysis includes typical use cases for multicore systems and their compatibility to these architectures. It has a strong focus on the platform's suitability for mixed-critical workloads with some cores executing safety-critical, hard-real-time applications. This paper proposes a novel shared memory protection unit to efficiently enforce spatial separation of the shared memory among the cores. Preliminary synthesis results are provided along with latency considerations relevant for hard-real-time application.**

I. Introduction

In several embedded domains there is a trend to reduce the number of electronic control units (ECU) by integrating more functionality into single ECUs. This requires higher performance from these ECUs. In the past, ever increasing performance of computing platforms was provided by increasing the clock speed and the amount of Instruction Level Parallelism exploited. This is no longer the case, since this development hit a point where the thermal power density becomes infeasible (Power Wall). The only remaining option for increasing the available computational power per chip is to distribute processing elements over it. This gives raise to an increased need to apply multicore architectures in all segments of the market. Certification standards such as IEC 61508 [1] and its domain specific derivatives DO-178b [2] or ISO 26262 [3] require that a failure in one application cannot influence the behaviour of another. This becomes more important in mixed-critical systems, where some applications might have a higher failure probability then others and thus might reduce the overall system integrity. Current interface standards like ARINC 653 Part 1 [4] for avionics or AUTOSAR for automotive reflect this by demanding "freedom of interference", or "separation" of applications.

Since they are easy to program and have high-performance most multicore platforms implement some sort of shared memory architecture. For low core counts bus-based systems provide the highest performance. Specifically in safety-critical systems, multicore systems are only slowly employed and core counts are very low. Therefore, we focus on such systems.

While shared memory architectures are convenient to use, shared resources constitute a problem, when it comes to separation. E.g., in shared-memory architectures with no protection, each application can access and manipulate other applications' data. This allows erroneous applications to modify the memory content and thus the behaviour of other applications.

In many single core systems CPUs have local memory protection units (MPU) or memory management units (MMU) that prevent processes from inferring with each other. MPUs are used to restrict access to predefined memory regions, usually configured by the operating system. The MPU verifies that the currently running process accesses only the address ranges it is eligible to access. If a process context switch takes place, this configuration is updated by the operating system. Instead of MPUs, MMUs can be used. Here, each process has its own virtual address space. When the process accesses a virtual address, the MMU looks it up in the processes translation table and translates it to a physical address. The MMU translation tables are located in the memory. On a context switch, the OS writes the address of the next processes translation table into the MMU. MMUs use a translation lookaside buffer (TLB) where table entries are cached to speed up address translation. The TLB is usually emptied on a context switch. A MMU can also be used to restrict access to some regions of the memory, because not all physical pages have to be made available to the virtual address range at the same time. Because a MMU can provide all processes with an individual virtual address space, applications don't have to run in specific memory locations. This results in more flexibility and speeds-up memory accesses in comparison to a software implemented memory management. The disadvantage of MMU utilization is that their timing behaviour is generally unpredictable [5].

In the context of this paper the term memory access control units (MAU) will be used as a generalization of MPU and MMU. MAU configuration is usually performed by operating system tasks running in privileged mode. In some current multicore architectures the cores are also equipped with individual MAU. These are then managed by the local operating system and ensure that all processes access their private data exclusively. If on such a platforms two cores run different operating systems (or two instances of the same OS), they have to trust each other. Nothing ensures that their configuration is correct in general.

These settings allow an OS running in privileged mode to bypass memory protection. Thus, we argue that an additional

978-1-4673-2684-1/12/$31.00 ©2012 IEEE

memory protection has to be established closer to the shared memory. Therefore, we propose to employ an additional (central) MAU to guard the shared memory. This module receives all requests, checks the address ranges, and access types.

We argue that the enforced spatial separation of shared memory by an additional centralized MAU will greatly simplify certification of multicore systems running multiple safety-critical applications or even mixed-critical workloads. In a first step, it has to be shown, that the separation of the applications works and fulfils all requirements. Afterwards, each application can be evaluated individually, because the first step has shown that there are no influences by other parts of the system.

There are also several combinations of local and shared MAUs conceivable. We discuss advantages and disadvantages of these concepts to identify architectures suitable for mixed-critical workloads with some cores executing safety-critical, hard-real-time applications.

Although, it seems to be a simple problem at first glance, to our knowledge we are the first to systematically discuss variants of MAU application in multicore processors and their impact on certification aspects.

In the following Section we introduce related work on this area. In Section 3 we discuss the advantages and disadvantages of various architectural concepts using local and/or shared MAUs. Section 4 gives a short overview of our preliminary results. Section 5 concludes this paper and Section 6 gives outlook to further work.

II. Related Work

A lot of current operating systems provide virtualisation features. These usually allow running multiple applications on the same core as if they were running in isolation. The applications are separated by the operating system. Usually the operating system uses MMU or MPU to ensure this. Violations to the separation are handled by the OS. The operating system is a trusted resource in this case. There are no mechanisms that handle consequences of a failing operating system. Samples of operating systems with virtualisation are SYSGO PikeOS [6], PharOS [7], Wind River Hypervisor [8], and Green Hills INTEGRITY Multivisor [9].

[10] proposes a compiler approach introducing a timing predictable form of paging, in which page-in and page-out points are selected at compile-time.

Many architectures avoid the problem of separable shared memory by avoiding shared memory altogether. One example here is the commercially available XMOS architecture [11], where cores communicate through crossbar switches.

In [12] a hardware/software system is presented which enables restriction of memory accesses and control flow of applications to protected domains within the address space with minimal architectural extensions to the processor core. Its applicability is also restricted to process separation in single core processors.

[13] uses an FPGA to monitor memory accesses. Erroneous accesses are detected and reported to the operating system. While effectively enabling fail-safe implementations, this approach does not provide separation and thus cannot be used for fail-operational applications.

A dynamic memory management system using a shared MMU is described in [14]. This approach focuses on supporting the OS in dynamic memory management and can be used for soft real time systems, while we target safety-critical, hard-real-time applications.

There is a body of work on memory management in networks-on-chip (NoCs). The focus of these works is in general management of shared memory and does not focus on applicability in hard-real-time systems or safety-issues. Due to their assumption of NoC communication they focus on packet transmission of data, not on separation of word accesses to the memory. E.g. [15] proposes a network-on-chip based shared memory architecture that uses a shared hardware MMU, while [16] focuses on security issues in shared memory NoCs.

ARMs TrustZone allows to switch cores dynamically into a secure mode. The TrustZone Address Space Controller restricts the access to some parts of the memory to cores in the secured mode. The solution focuses on security and its use for safety is limited. Because TrustZones can only separate between secure and normal mode[17] only one safety critical application in the system is possible without support of operating systems or virtualisation.

For safety-critical real time systems not only spatial separation is needed. Since multiple cores are competitively accessing the memory, temporal separation is needed to ensure real-time properties. Temporal separation needs to be taken care of in practically all parts of the hardware system and a large body of work exists. [18] proposes a fully deterministic architecture with repeatable timing. The propeller architecture is a commercially available multicore processor implementing a round robin arbitration scheme on the bus connecting shared resources to cores [19]. In [20] a superscalar in-order processor is enhanced to provide hard real time capability. For temporally separated multicore systems, NoC approaches are often utilized. E.g. in [21] a statically scheduled time-triggered Ethernet is employed. [22] uses a very similar approach, while [23] utilizes globally synchronized frames to provide QoS guarantees. Another approach is arbitrating shared memory buses in a predictable fashion (e.g. [24]).

III. Architectural concepts

As already discussed in Sec. I there are two basic concepts providing hardware support for memory protection: The very lightweight MPU on the one hand and the MMU which additionally supports memory virtualisation on the other hand.

In a shared memory multicore architecture these MAU components can be placed either locally in the processing elements or centralized at the shared memory. Also combinations of both provide an added value and exhibit new properties.

This results in eight different architectural variants depicted in Fig. 1. All of these concepts provide some sort of memory

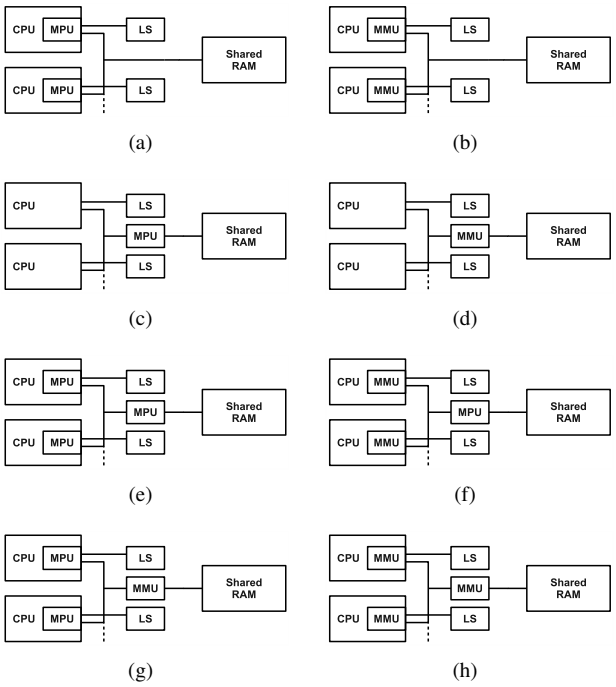

Fig. 1. Architectural Concepts (LS = Local Storage)

protection. Nevertheless, varying capabilities favouring utilization in different scenarios result from the design choices made here. These are discussed in the following.

A. Local MAU

Local MAU implementations as depicted in Fig. 1(a), 1(b), 1(e) - 1(h) allow protection of private memory (local storage - LS) such as scratch pads. Operating systems can use them to control access of their processes to the private memory. These concepts are interesting for systems where multiple processes run on the same core managed by an operating system.

Concepts with local MAUs only (Fig. 1(a), 1(b)) will also provide protection of memory regions of other applications to a certain degree. Nevertheless, the instance configuring the MPUs (usually an operating system instance running on the core) will have to be trusted not to interfere with memory locations not assigned to this core. In this case it can be assumed that the memory protection is always configured correctly. This obviously constitutes a safety leak in general.

B. Shared MAU

A shared MAU as displayed in Fig. 1(c) - 1(h) can enforce spatial separation between cores in hardware. If configured accordingly, no core can access other cores' memory locations (unless this is explicitly enabled). Since the cores are separated by a dedicated unit, systems using these architectures can in principal be certified modularly. Once it is proven, that the separation is guaranteed, each core can be evaluated individually. These concepts also provide security features. A core that has been compromised by an intruder cannot access the private data of other cores, if the configuration of the shared MAU forbids this.

Architectures with only shared MAU (Fig. 1(c), 1(d)) are not able to protect memory which is local to the core and private to a process running on this core from unauthorized access by other processes running on the same core.

Nevertheless, they are a resource saving alternative for systems that don't need memory protection within one core. E.g. systems with no operating system, where each core runs only one application on bare metal.

C. Local and shared MAU

Systems providing both, local and shared MAU (Fig. 1(e) - 1(h)) are very flexible and can be used in most applications. The local MAU of each core protects the private memory and can be used to protect the tasks of one core from each other. Separation of the cores is done with the shared MAU. It ensures that the OS and tasks of each core do not access other core's memory, unless it is intended by the system integrator. In cases where not all of these units are needed they can remain unused. Typical use cases for such architectures are systems running multiple applications on the same processor. Each core can have a different operating system and use the local protection unit for their tasks.

D. Memory management units

While both types of MAU provide hardware support for memory protection, memory virtualisation is not supported by MPUs. For this MMUs need to be utilized.

A local MMU (Fig. 1(b), 1(f), 1(h)) enables the system not only to protect the private memory, but also to virtualise the private memory to the processes [25]. It translates a virtual address of the application either to a shared memory address or a private memory address, depending on where the data is located. This simplifies the implementation of local buffering techniques and increases the applications performance. The operating system in such scenarios needs to copy the data between the shared and the private memory. It also configures the processes translation table with respect to the current data location. A software cache can also benefit from local MMU, because the address translation will be performed in hardware. Within a core, a MMU is usually tightly coupled to the core's internal architecture to perform better.

A shared MMU as shown in Fig. 1(d), 1(g), 1(h) can be used in systems with no need for virtualisation of local storage.

The total size of the translation lookaside buffer (TLB) and the translation tables will remain the same for both, shared and local MMU. A reduced size of the TLB will reduce the systems performance. If temporal separation is needed, the TLB has to be separated into regions for each core. Otherwise one core will overwrite other cores' entries and thus prolong their memory access time. Local MMUs can be handled by current operating systems, which reduces implementation effort and the update mechanisms are known.

The use of a MPU instead of a MMU has the advantage, that MPUs have constant latency enabling WCET analysis. MMU translation tables are stored in the memory, therefore the latency of a MMU translation depends on the status of the

translation lookaside buffer (TLB) and the access latency of the memory [5], [26].

Placing a MMU in both, the processing element and the shared RAM (Fig. 1(h)) does not provide benefit with respect to flexibility. The translation tables of both MMUs can be combined into a single table located in the local MMU of each core. Combining the translation table has the advantage that the address translation is done by only one MMU. This reduces the latency for the memory access and saves the resources of one of these MMU. Thus, the concepts of Fig. 1(h) and 1(f) provide identical features. Therefore we reject the application of shared MMUs.

IV. Preliminary Results

We have implemented a first prototype of a shared MPU separating memory accesses of Altera Nios II Cores on a Stratix III FPGA Development Board.

Here, it utilizes 6796 combinational logic cells. 30 % of them are used to check the addresses of accesses, 20 % are used for the bus handling. The remainder is used for control registers and interrupt generation and handling.

To drive it at 125 MHz frequency one register bank is needed. Thus, the latency introduced by the MPU is one clock cycle. It supports burst and pipelined accesses. Therefore the influence on the throughput of the system will be negligible.

V. Conclusion

We presented an overview on the use of memory protection techniques in bus-based, shared-memory systems.

From the previous discussion, we conclude that a central MPU is a lightweight solution providing spatial separation between cores. It allows WCET analysis to result in tight time-bounds for hard-real-time applications, since it introduces only a constant delay. We therefore chose this variant for implementation. This choice does not come at any disadvantage, as cores which implement functionality not requiring WCET analysis, but will benefit from memory virtualisation, can be extended with MMUs. When connected by a time separating bus system this enables execution of highly-critical tasks requiring WCET analysis on some cores concurrently to less critical tasks with full benefit of memory virtualisation on others.

VI. Further Work

Currently we are optimizing the central MPU implementation with respect to size. Subsequently, we will provide an implementation of a shared MMU as a reference system for size, throughput, latency and energy comparison. In the future we will also investigate in configuration and reconfiguration schemes and modules of those shared memory access control units (MAU). The integration of shared MAU configuration into operating system and tool chains will be an important part of our work. In a latter step configurations for MAUs will be inferred from application models.

Caches also influence the behaviour of such a system. A cache in front off the shared MAU can delay the detection of a violation. Cache snooping protocols might even forward unchecked data to other cores. We will focus on this and provide solutions for this problem.

Concurrently, we are also investigating in temporal separation schemes and WCET estimation techniques for architectures with shared SDRAM. These techniques will be combined with the spatial separation approaches into a bus-based, shared-memory platform for mixed-critical systems.

Acknowledgment

This work was partially funded within the ARTEMIS project grant RECOMP under the BMBF research grant with funding ID 01IS10001K.

References

[1] *IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems*, IEC Std.
[2] *DO-178B Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics Std.
[3] *ISO 26262 Road vehicles – Functional safety*, ISO Std.
[4] *ARINC 653-2 Avionics Application Software Standard Interface Part 1 - Required Services*, Aeronautical Radio, Inc. Std., December 2005.
[5] X. Zhou and P. Petrov, "The interval page table: virtual memory support in real-time and memory-constrained embedded systems," in *SBCCI '07*. New York: ACM, 2007.
[6] "SYSGO PikeOS."
[7] C. Aussaguès *et al.*, "PharOS, a multicore OS ready for safety-related automotive systems: results and future prospects," in *Embedded Real Time Software and Systems 2010*, May 2010.
[8] "Wind River Hypervisor."
[9] "Green Hills INTEGRITY Multivisor."
[10] I. Puaut and D. Hardy, *Predictable paging in real-time systems: a compiler approach*, 2007.
[11] "XMOS." [Online]. Available: http://www.xmos.com/
[12] R. Kumar *et al.*, "A system for coarse grained memory protection in tiny embedded processors," in *DAC '07*. New York: ACM, 2007.
[13] N. Ho and A.-V. Dinh-Duc, "MemMON: run-time off-chip detection for memory access violation in embedded systems," in *SoICT '10*. New York: ACM, 2010.
[14] M. Shalan and I. Mooney, V. J., "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," in *CODES 2002*, 2002.
[15] M. Monchiero *et al.*, "Exploration of Distributed Shared Memory Architecturesfor NoC-based Multiprocessors," in *SAMOS 2006*, 2006.
[16] L. Fiorin *et al.*, "Secure memory accesses on networks-on-chip," *IEEE Trans. Comput.*, 2008.
[17] ARM, "ARM Security Technology - Building a Secure System using TrustZone Technology," White Paper.
[18] B. Lickly *et al.*, "Predictable programming on a precision timed architecture," in *CASES '08*. New York, NY, USA: ACM, 2008.
[19] "Propeller." [Online]. Available: http://www.parallax.com/propeller/
[20] J. Mische *et al.*, "How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT," *ARCS 2010*, 2010.
[21] R. Obermaisser and B. Huber, "The GENESYS Architecture: A Conceptual Model for Component-Based Distributed Real-Time Systems," in *SEUS '09*. Berlin, Heidelberg: Springer-Verlag, 2009.
[22] A. Jantsch, "Models of Computation for Networks on Chip," in *ACSD '06*. Washington, DC, USA: IEEE Computer Society, 2006.
[23] J. W. Lee, M. C. Ng, and K. Asanovic, "Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks," in *ISCA '08*. Washington, DC, USA: IEEE Computer Society, 2008.
[24] H. Shah, A. Raabe, and A. Knoll, "Bounding wcet of applications using sdram with priority based budget scheduling in mpsocs," in *DATE 2012*, 2012.
[25] H. Cho *et al.*, "Dynamic data scratchpad memory management for a memory subsystem with an MMU," in *LCTES '07*. New York, NY, USA: ACM, 2007.
[26] M. D. Bennett and N. C. Audsley, *Predictable and Efficient Virtual Addressing for Safety-Critical Real-Time Systems*, Washington, 2001.