Towards Temporal and Spatial Isolation in Memory Hierarchies for Mixed-Criticality Systems with Hypervisors

Bekim Cilku and Peter Puschner
Institute of Computer Engineering
Vienna University of Technology
A1040 Wien, Austria
Email: {bekim, peter}@vmars.tuwien.ac.at

***Abstract*—In mixed-criticality systems, applications with different levels of criticality are integrated on the same computational platform. Without a proper isolation of the different applications of such a mixed-criticality system certification gets expensive, because it has to be shown that application components of lower criticality do not hamper the correct operation of the critical applications. Therefore, all components – even the less critical ones – have to be certified for the highest criticality level. The use of hypervisors promises to shield applications of different criticality from each other, thus keepting certification cost reasonable. Indeed hypervisors can provide spatial isolation. Further they can prohibit certain types of temporal interference. We show, however, that full temporal isolation is only achievable if the hypervisor is run on appropriate hardware. We discuss hardware mechanisms that thwart respectlively facilitate temporal isolation. This way we provide a guideline for the sharing of resources and the realization of memory hierarchies for mixed-criticality multi-core systems.**

***Keywords*—mixed-criticality systems; partitioning; multi-core; isolation; time predictability; memory hierarchy;**

I. Introduction

In the past, a single function had a dedicated embedded unit which lead to highly distributed systems. The drawback of these systems is the large number of embedded units and the high number of connectors. Today, due to the higher processing capability that multi-core embedded systems have reached there is an increasing trend towards the integration of multiple applications in a single shared hardware platform [1]. Some of the integrated applications can have firm real-time constraints, demanding a formal proof that the deadlines are met, while the others can be less restrictive. For such a mixed-criticality system, only the set of critical applications is a subject of certification and the rest of the applications do not need to be certified or can be certified to a lower level [2]. Obtaining a certification only for critical applications can become a difficult work due to the hardware sharing dependency and the diversity of functionalities that are performed concurrently in the system. The key approach to reduce complexity is to prevent interference between integrated applications in the temporal and spatial domain [3].

An approach to increase maintainability and to avoid the growing validation and certification effort for critical application in mixed-criticality systems is to run applications in virtual partitions on top of a hypervisor. This approach on one side requires fewer processors, reduces the cost of embedded systems, and hides heterogeneous hardware transparently and on the other side allows applications with different time requirements to be designed and verified by offering them isolated environments. However, establishing temporal isolation between virtual partitions is one of the major issues. Even if the virtual partitions are logically independent they still share resources (e.g., CPU, cache, bus, memory), making their temporal behavior inter-dependent.

The goal of this paper is to identify the sources of interference in the memory hierarchy for mixed-criticality systems with hypervisors and also to present a memory architecture that mitigates temporal interference between critical partitions. The solution provides composability and predictability for critical virtual partitions, which helps to bound WCET, to reduce complexity, to simplify the verification process, and to offer the best performance for non-critical partitions without interfering with the time of critical ones.

The paper is structured as follows. The following section defines temporal and spatial isolation in mixed-criticality systems with a hypervisor. Section three identifies sources of interference in memory hierarchies and gives a solution to mitigate this effect. In section four we review the related work. The paper finishes with a conclusion in section 5.

II. Temporal and Spatial Isolation

Spatial isolation. Virtual partitions are spatially isolated if the code or private data of one partition cannot be altered by any other partition [4]. In a mixed-criticality system with a hypervisor, spatial isolation is achieved if each partition has its own address space on the shared memory. This can be done by address translation and assigning a different memory location to each virtual partition. A naive solution is to equip the hardware with a Memory Management Unit (MMU) to translate virtual addresses of the virtual partitions to the global address space. MMU tables protect the assigned address space of one partition from violations by the other partitions [5].

Temporal isolation. Temporal isolation ensures that a virtual partition cannot affect the ability of the other virtual partitions to access a shared resource [4]. Establishing temporal isolation between virtual partitions is a major issue considering that the main characteristic for an integrated architecture is sharing computational, communicational, and

memory resources. Even if the virtual partitions are logically independent they still share resources making their temporal behavior inter-dependent. In the following we identify the main sources of temporal interference in memory hierarchies (cache, bus and DRAM) and show solutions to mitigate them.

III. Memory Hierarchy for Virtualized Mixed-Criticality Platform

A. Cache memories

Cache memories are small high speed memories used in computer systems to temporarily hold those parts of the main memory which are (believed to be) currently in use. Data located in the cache memory are accessed in much less time than those located in main memory. This success relies on the principle of temporal and spatial locality that a program has during its execution [6].

In a virtualized multi-core platform with private caches, virtual partitions that are sharing a core will also share its cache. Cache accesses of different partitions that run on the same core are interleaved. Thus, the temporal behavior of one virtual partition depends on the cache accesses performed by other virtual partitions. To avoid such interference between execution environments (virtual partitions), cache partitioning has been proposed. This means that the cache is split into segments, and each execution environment is forced to use only the segments which it owns. Therefore all the cache data of one execution environment are protected from evictions due to the memory accesses of others. Cache partitioning strategies can be implemented by adding additional hardware to the cache (hardware approach) [7] or with support of a compiler or operating system (software approach) [8] [9]. Using fixed hardware-based cache partitioning limits the flexibility of the cache partitioning approach. Software based partitioning is more flexible considering the number of partitions that can be created in one core. Also the size of cache partitions for different virtual partitions can be variable (Figure 1). However, the use of this technique increases the overhead of task execution.

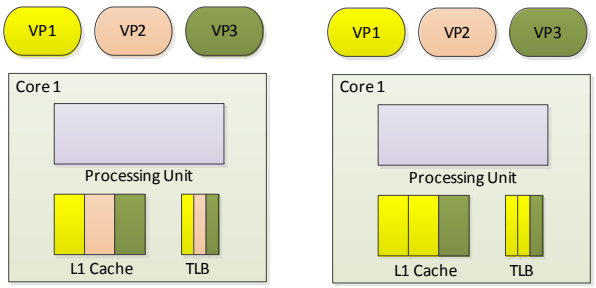

Fig. 1: Cache and TLB partitioning.

A source of non-concurrent interference are also tagged TLBs. After context switching, depending on the behaviour of the other executing partitions, TLB entries might or might not be present when a partition executes again. This results in different memory access times and hence temporal interference. Implementing TLB partitioning would eliminate TLB context switching dependencies between virtual partitions.

B. On-chip communication

Components that share the same memory are often interconnected through a bus. A bus is a medium that transfers data between components and shared memory according to a defined policy in the arbiter [10]. Each component has a *request line* to inform the arbiter of its needs for bus access, and a *grant line* to receive permission to use the bus. When the bus is available, based on the arbitration algorithm, the arbiter grants access to the bus to one of the virtual partitions [10]. For critical partitions it is important to satisfy bandwidth and response time requirements. Thus, without further adaption, bus transactions of different virtual partitions can interfere with each other. The easiest way to achieve temporal isolation between partitions is by adding a global notation of time to a static arbitration scheme, where each partition can access the memory in an a-priori determined time slot [11]. However, this solution is too restrictive for mixed-criticality systems because all virtual partitions must be considered at the same level of criticality.

In the following we discuss a bus arbitration policy that guarantees temporal isolation for critical partitions and higher transaction performance for non-critical ones. The proposed arbitration scheme is a two-level policy that combines a static with a dynamic arbitration scheme.

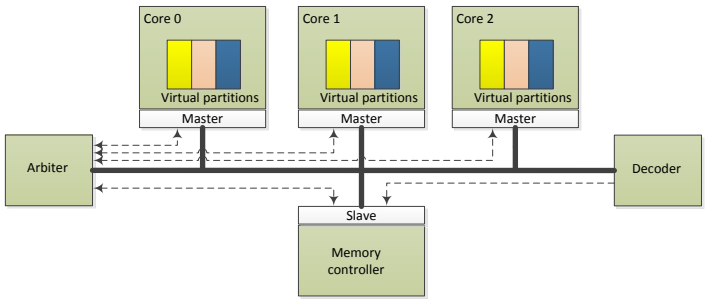

Fig. 2: Bus-based communication.

The static priority scheme is the main bus access controller. It is based on TDMA with equal-length time slots. The length of the slot is the worst-case time that a single request needs to be serviced. Each critical partition has a-priory assigned time slots. Whenever the partition generates a request for a memory access it must wait for its related time slot. A virtual partition has to issue a bus permission requests before its time slot starts, otherwise it has to wait for its next dedicated time slot. If it would be allowed to serve a request in the middle of a static time slot the data transfer could overlap with the time slot reserved for the next access. Also, a preemption of a critical transaction is not allowed. Individual partitions that

need higher bandwidth can be assigned multiple slots making bandwidth division asymmetric. For example, in Figure 3 it can be seen that if one core with a critical partition (in this example Core1) needs to access memory more frequently it can be scheduled more often by the arbiter. By knowing the time of one slot and their frequency in one round of arbitration we can derive the bandwidth and the maximum latency for each critical partition, which makes the system predictable.

To provide high performance for non-critical partitions without affecting critical ones, we propose dynamic arbitration to schedule their requests. The dynamic arbitration is built on top of the static one (Figure 3). The activation and deactivation of this policy is controlled by the static policy. A few sequential time slots from static arbitration are merged and reserved for non-critical applications. During these time intervals, dynamic arbitration is activated and all non-critical partitions are competing for transactions. At runtime the arbiter schedules the requests based on the dynamic policy. To ensure that the dynamic arbitration cannot interfere with the static arbitration, an additional rule has to be enforced. The dynamic arbiter must not serve any request that was issued after the start of the last timeslot in the sequence of dynamic time slots. Otherwise, this request could overlap with the next static time slot for the critical cores. Dynamic arbitration provides better performance for non critical applications but the worst-case completion time of requests cannot be bounded.

C. DRAM memory

Multi-core systems are mostly implemented with shared main memory to achieve a good balance between performance, cost and power consumption [12]. Main memory is usually based on DRAM technology using single transistor-capacitor pairs called cells, which are organized in a two-dimensional structure of rows and columns, creating arrays [13]. The access time of DRAM bit cells is highly variable as a result of the time dependence from previously issued accesses (e.g., same/different row, same/different bank, read/write, refreshing). This makes the DRAM technology prone to temporal interference. The first solution to eliminate this effect is to add the maximal delay time to each memory access slot by including all possible delays. This approach however increases the worst-case latency for memory accesses.

In the following we describe some approaches that decrease memory latency and stabilize the access time. When a memory row is accessed it must first be copied into the row buffer and then the data are transfered to the memory controller. The two primary row-buffer management policies are the open-page policy and the closed-page policy [13]. An open-page policy introduces temporal interference because depending on the previous request the row can be accessed immediately, if the sequential accesses are on the same row, or the row has to be closed first and then the new row to be loaded into the row buffer. To reduce the access time variability we propose the use of closed-page policy. For this policy, each access of the row is loaded to the buffer, accessed and immediately closed.

Also, bank partitioning is a way to reduce memory access time by pipelining the memory access. This can be done assigning a separate bank to each partition. If the number of virtual partitions in the system is at most as large as the number of banks in the memory it is possible to ensure that every partition accesses always the same bank which prevents all other partitions from changing that bank's state.

The small capacitors used in DRAM discharge over time due to power leakage. So to preserve the value, the capacitors must be refreshed periodically. The main issue with the refreshing process is that data accesses are asynchronous to memory refreshes. Thus, refreshing can delay a data access if the corresponding row is already in the process of refreshing. In DRAMs where refreshing is programmable we can schedule the refreshing process in the memory controller by giving it a separate time slot in the schedule. During this time slot no transaction is allowed on the bus, and the memory controller just sends the refreshing command. Thus, memory accesses and refreshes cannot collide.

TABLE I: Mechanisms to prevent temporal interference in memory hierarchy

Component	Mechanism for temporal isolation
cache	hardware/software cache partitioning
bus	static arbitration for critical partitions
off-chip memory (DRAM)	closed-page policy, programmable refreshing

IV. Related Work

There have been a few studies on building memory hierarchies for mixed-criticality systems that support WCET analysis.

Akesson et al. [12] presented a memory controller for SDRAM called *Predator* that provides a guaranteed minimum bandwidth and maximum latency bound. The controller combines elements of statically and dynamically scheduled memory controllers. Command operation times for read or write are fixed and can be precomputed at design time. They obey all SDRAM timing constraints and have fixed latencies. Arbitration uses credit controller static priority (CCSP) scheduling to regulate the rate of requests and to guarantee maximum bandwidth. This arbitration policy allows decoupling between latency and rate for each client. All hard real-time responses are delayed to their worst-case latency to provide temporal isolation between requesters.

Paolieri et al. [14] built a memory controller that enforces the upper-bound delay for critical tasks to not be violated. This feature makes the controller timing-analysable. The arbitration is split into two levels, one that schedules requests among different cores (inter-core bus arbiter) and the other one that schedules thread-requests from the same core (intra-core bus arbitration). Critical tasks are always run with higher priority than non-critical tasks. The WCRT is calculated by using the maximum possible time needed for a single transfer multiplied by the number of possible other critical tasks that can generate requests at that moment.

The approach of Reineke et al. [15] for a DRAM controller for the PRET machine is based on a multi-thread concept. The processor implements a four thread-interleaved pipeline and a

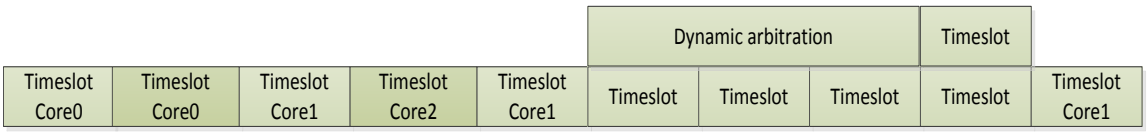

Fig. 3: Dual-policy arbitration scheme.

separate bank is assigned to each thread. Accessing DRAM in this way removes the conflicts between threads considering banks as a non-shared memory and also reduces the WCET by interleaving bank accesses. The arbitration is simple round-robin.

Lakis et al. [16] analyse time parameters of DRAM transactions and minimal separation between transactions by defining the maximun time of transactions for different directions (read after write, write after read, different burst length). To increase the bandwidth they also include interleaving transactions in a single data rate (SDR) controller.

V. Conclusion

Applications with different time criticality use different design and verification methods. Using a hypervisor with the properties of temporal and spatial isolation in mixed-criticality systems makes it possible that critical applications are designed and verified in isolation.

Our proposed hardware architecture helps a hypervisor to guarantee temporal isolation for each virtual partition when they access shared DRAM memory. This makes the memory hierarchy composable and therefore suited for mixed-criticality integration. By partitioning the private caches, unpredictability due to context switches is removed, and determining the temporal characteristics of the whole system is made easier. By using a static bus arbitration policy, interference due to concurrent accesses of critical partitions is prevented. Combining the static arbitration with a dynamic arbitration policy we gain additional efficiency for non-critical partitions while still retaining composability for the critical ones. Furthermore interference due to the state of the DRAM is eliminated within each timeslot. This makes the proposed memory hierarchy suited for mixed-criticality integration.

Acknowledgment

This work has been supported in part by the European Community's Seventh Framework Programme [FP7] under grant agreement 287702 (MultiPARTES).

References

[1] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, 2010, pp. 13–22.

[2] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, 2011, pp. 34–43.

[3] B. Motruk, J. Diemer, R. Buchty, R. Ernst, and M. Berekovic, "Idamc: A many-core platform with run-time monitoring for mixed-criticality," in *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*. IEEE, 2012, pp. 24–31.

[4] B. Leiner, M. Schlager, R. Obermaisser, and B. Huber, "A comparison of partitioning operating systems for integrated systems," in *Computer Safety, Reliability, and Security*. Springer, 2007, pp. 342–355.

[5] R. Obermaisser and B. Leiner, "Temporal and spatial partitioning of a time-triggered operating system based on real-time linux," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, 2008, pp. 429–435.

[6] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, pp. 473–530, 1982.

[7] D. Kirk, "Smart (strategic memory allocation for real-time) cache design," in *Real Time Systems Symposium, 1989., Proceedings.*, 1989, pp. 229–237.

[8] F. Mueller, "Compiler support for software-based cache partitioning," in *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, ser. LCTES '95. New York, NY, USA: ACM, 1995, pp. 125–133. [Online]. Available: http://doi.acm.org/10.1145/216636.216677

[9] J. Liedtke, H. Haertig, and M. Hohmuth, "Os-controlled cache predictability for real-time systems," *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, vol. 0, p. 213, 1997.

[10] S. Pasricha and N. Dutt, *On-chip communication architectures: system on chip interconnect*. Morgan Kaufmann, 2010.

[11] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[12] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable sdram memory controller," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '07. New York, NY, USA: ACM, 2007, pp. 251–256. [Online]. Available: http://doi.acm.org/10.1145/1289816.1289877

[13] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[14] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for wcet analysis of hard real-time multicore systems," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 57–68.

[15] J. Reineke, I. Liu, H. Patel, S. Kim, and E. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference on*, 2011, pp. 99–108.

[16] E. Lakis and M. Schoeberl, "An SDRAM controller for real-time systems," in *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013. [Online]. Available: http://www.jopdesign.com/doc/sdramctrl.pdf