Software-enforced Interconnect Arbitration for COTS Multicores

Marco Ziccardi, Alessandro Cornaglia, Enrico Mezzetti, and Tullio Vardanega

University of Padova
{mziccard, acornagl, emezzett, tullio.vardanega}@math.unipd.it

Abstract

The advent of multicore processors complicates timing analysis owing to the need to account for the interference between cores accessing shared resources, which is not always easy to characterize in a safe and tight way. Solutions have been proposed that take two distinct but complementary directions: on the one hand, complex analysis techniques have been developed to provide safe and tight bounds to contention; on the other hand, sophisticated arbitration policies (hardware or software) have been proposed to limit or control inter-core interference. In this paper we propose a software-based TDMA-like arbitration of accesses to a shared interconnect (e.g. a bus) that prevents inter-core interference. A more flexible arbitration scheme is also proposed to reserve more bandwidth to selected cores while still avoiding contention. A proof-of-concept implementation on an AURIX TC277TU processor shows that our approach can apply to COTS processors, thus not relying on dedicated hardware arbiters, while introducing little overhead.

1998 ACM Subject Classification D.4.7 Organization and Design/Real-time systems and embedded systems

Keywords and phrases Multicore, Resource Arbitration, Interference, Mixed-Criticality

Digital Object Identifier 10.4230/OASIcs.WCET.2015.<first-page-number>

1 Introduction

An extraordinary growth in the complexity of software systems has occurred in the last decades. This complexity growth also entailed an increase in the computational demand [6] that could only be sustained by the adoption of advanced and powerful multicore and manycore systems, which have become the de-facto reference standard for computing platforms. This unrelenting transition to multicore systems also involves the application domain of real-time embedded applications (avionics, automotive, aerospace, etc.), where predictability and analyzability in the time domain are stringent requirements. The coexistence of multiple applications running in parallel on distinct cores in the same platform, however, complicates the analysis of the worst-case execution time (WCET) behavior of programs running on such systems, due to the inter-core timing interference caused by contention on access to shared hardware resources. The WCET, in turn, is the main input to schedulability analysis, which is a mandatory test for real-time systems to guarantee that tasks will meet deadlines at run time.

Notable effort has been devoted in the last few years to analyze the WCET behavior of systems deployed on modern multicore platforms. Two main research paths have been followed to cope with the inter-core interference problem [4]. One class of approaches [5, 8, 14, 17] aims at precisely analyzing the timing interference to provide safe and tight bounds, to be fed into schedulability analysis as an additive factor. However, besides the inherent complexity of deriving trustworthy bounds on complex architectures, the worst-case inter-core interference thereby computed is likely to be too large to be usable. The second class of

© M. Ziccardi, A. Cornaglia, E. Mezzetti and T. Vardanega;
licensed under Creative Commons License CC-BY
15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015).
Editor: Francisco J. Cazorla; pp. 1–10

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

approaches [9, 16, 18, 21] attacks this problem by carefully limiting/controlling the sources of inter-core interference in the system to make it more analyzable. This can be achieved by partitioning the shared hardware resources in a system either spatially (physical partitioning) or temporally (e.g., through a hypervisor or arbitration schemes). Resource partitioning is particularly interesting when dealing with mixed-criticality systems (MCS) [20] where the computational power guaranteed by multicore platforms allows hosting on the same target applications that are classified at different criticality levels, according to the applicable certification and qualification standard [13]. In the MCS scenario, we need to avoid high criticality tasks to be interfered by lower criticality ones in a possibly uncontrollable way.

Time and space partitioning are well-known concepts also in singlecore settings and are advocated, for example, by the ARINC-653 [2] standard to provide isolation across applications running on the same core. On a multicore architecture, however, interference not only occurs between tasks running on the same core but can also arise when tasks on different cores access platform resources (e.g. shared memory, cache), often through a shared interconnect. Concurrent accesses of different criticality applications to a shared interconnect may heavily affect the performance of high criticality ones. This performance impact is often complex to quantify as it depends on the number of applications running in parallel as well as on the specific hardware arbitration policy employed. A task spending only 10% of its time fetching shared memory may suffer up to 300% interference on an eight core processor [15].

To mitigate the problem of interference and achieve analyzability, statically arbitrated interconnects are often used. Time division multiple access (TDMA) policy, amongst other, allows ruling out inter-core interference by means of temporal isolation [17]. However, the hardware support provided in Commercial Off The Shelf (COTS) multicores to arbitration policies, regulating accesses to shared interconnects, is typically oriented to achieve better average performance rather than system analyzability. Relying on specific hardware-enforced policies is, therefore, not always possible without recurring to (often unaffordable) custom hardware solutions. In this paper we propose an approach to the arbitration of hardware shared resources that relies solely on system software, without posing particular requirements on the hardware platform. Our solution applies to partitioned multicore systems where inter-core communication is regulated by the real-time operating system (RTOS). This is the case, for example, of automotive systems abiding by the OSEK/AUTOSAR [3] standards where a specific inter-core message-passing API is provided. Our approach consists in deploying a flexible software-arbitration layer through a specific API implementation, while, of course, preserving the API semantics. We exploit the inherent flexibility of software approach to enforce both a *pure* TDMA arbitration policy and a bandwidth-reservation flavour of TDMA, which is better equipped to meet mixed-criticality application requirements. We show that our technique manages to control the interference by construction and provides timing guarantees for all accesses to the interconnect. We also show that our approach is *hardware agnostic* as it can be applied to complex interconnects employing either priority-driven or round-robin policies (or a combination of the two).

This paper is organized as follows: Section 2 contextualizes our work, Section 3 introduces both the base and bandwidth-reservation versions of our software-based TDMA. An experimental evaluation is given in Section 4. Finally, Section 5 draws some conclusions.

2 Related Work

The reduction of inter-core interference on shared resources is widely recognized as an enabler of timing analysis of multicore systems. Both hardware- and software-based solutions have

been considered as a means to achieve better analyzability without compromising performance. Shared interconnects, as a main source of inter-core interference, have captured the attention of the timing analysis community. From the timing analysis standpoint, inter-core interference is generally studied under the umbrella of Worst-Case Response Time (WCRT) analysis where the focus is set on using bounds on the inter-core interference in increasingly complex response time analysis equations. Analysis techniques for several arbitration schemes on COTS or custom hardware can be found in the literature, based on more or less complex models, such as arrival curves [17], event models [5] or timed automata [8]. Generally, however, the more complex the arbiter, the more pessimistic the WCRT bounds.

A classic solution to cope with contention is TDMA, where clients accessing a shared resource are served according to a fixed-time slots scheme. TDMA is widely used to manage shared interconnects on the account that it provides guaranteed bandwidth and predictable latency, regardless of the the number of contenders. Predictability, however, comes at the cost of a relative reduction in the interconnect utilization. A Round-Robin (RR) scheme can be used to improve on this aspect of TDMA: time slots are rotated only across active contenders. In comparison to TDMA, RR guarantees better utilization at the risk of being unfair to certain contenders, depending on specific access patterns [18]. An alternative approach to improve utilization consists in using a TDMA-based arbiter that varies core-to-slot allocation and slot size, and loads the *bus schedule* from memory [16]. Adaptive resource arbiters, such as FlexRay [1], have been recently studied in the context of mixed-criticality automotive software [10]. These arbiters combine static approaches as TDMA with dynamic ones, such as RR. Tough effective, hardware-based approaches are not generally implementable on COTS hardware and are inherently less flexible than their software-based counterparts.

Our approach is more related to software-based approaches, which allow abstracting away from the actual platform, and promise easy reconfigurability in response, for example, to run-time events. A software-enforced arbitration scheme for mixed-criticality systems is discussed in [21], where a memory throttling controller with variable budget assignment is exploited to limit the memory request rate of cores running non-critical tasks. This technique, however, is quite intrusive as it requires periodic checks of performance counters to monitor budget consumption. Moreover, critical tasks are only allocated to one processor and the number of supported interfering cores is limited. Our work, instead, does not pose any constraint in the assignment of critical tasks, and guarantees timing isolation regardless of the number of contenders. However, we do not account for all accesses to the shared interconnect, but only for those caused by inter-task communication. A Time-Triggered and Synchronisation-based (TTS) scheduling strategy that targets partitioned mixed-criticality systems with periodic task sets has been recently presented in [9]. TTS isolates tasks at different criticality levels and accounts for the effect of memory contention on task execution by relying on synchronization mechanisms (barriers) and fixed preemption points. Scheduling decisions are global, even though the algorithm enforced partitioning, which may incur non-negligible overheads [19]. Our approach relies on a relatively simple implementation and does not assume any specific task scheduling algorithm.

3 Software-enforced Resource Arbitration

It is not unusual for a typical embedded system RTOS to be responsible for inter-task and inter-core communication. Software specifications as, for instance, ARINC-653 [2] and AUTOSAR [3] offer a communication scheme based on message-passing abstractions: two tasks are not allowed to share memory, and all communication taking place between them

must be performed via messages sent through the RTOS API. The RTOS, in fact, exposes APIs for the creation of different kinds of communication channels (e.g. ports in ARINC-653 and Inter-OS Application Communication IOC buffers in AUTOSAR) and for their use, according to the respective semantics. The intuition behind our work is that, since the operating system is the mediator of all communication across applications, then it can also act as a request arbiter when such communication involves accessing a shared medium. We extend the RTOS implementation of the message-passing API to enforce a configurable TDMA arbitration policy: this is done by splitting exchanged messages into chunks and arbitrate each chunk transfer in TDMA-like time slots. TDMA frame, slots and chunk size can be configured to better meet the application requirements. To address specific needs of mixed-criticality systems, a more *flexible* TDMA-based arbitration is also proposed that allows reserving more than one communication slot to selected cores. This enhanced version of TDMA enables the provision of better bandwidth levels to cores running high-criticality applications.

The so-enforced software-based TDMA arbitration mechanism (SW-TDMA) has several advantages. First of all, the combination of message-passing API and TDMA enforces isolation between tasks running on different cores and makes the effects of bus contention completely predictable. Moreover, the adoption of software-based arbitrations is transparent to user applications as the arbitration mechanism is implemented at kernel level and does not affect either the syntax or the semantics of message passing APIs. Finally, the ability to reserve more than one arbitration slot to selected cores enables more complex arbitration policies to be designed without the need for expensive hardware. As an improvement over system flexibility, bandwidth levels can also change at run time in reaction to specific events or changes in the mode of operation. In the following we present our approach to enforce flexible TDMA-based arbitration schemes for inter-core communication on a shared interconnect. Before entering into details, we introduce our model and notation.

3.1 Assumptions and Model

We consider a multicore processor architecture with n cores $\{C_0, ..., C_{n-1}\}$, each one equipped with its own local memory (e.g. cache and/or scratchpad). Cores also share a global memory that can be accessed via a simple system bus or more complex interconnects as, for instance, crossbars. We assume that all cores share a common time source TS. We consider partitioned systems, where each application is statically assigned to one of the n cores. An application allocated to core C_i comprises a set of tasks (schedulable on C_i) and is granted exclusive use of a local memory area, separated from other applications, for storing its code and data.

Communication across tasks is only allowed through the RTOS, which becomes responsible for all memory transfers. To that extent, the RTOS API exposes a pair of procedures to read/write data from/to a communication channel shared across tasks: i) *receive_message*: copies a message (sequence of bytes) from a communication channel to the calling task's memory; ii) *send_message*: copies a message (sequence of bytes) from the task's memory to a communication channel. Inter-task communication between tasks running on the same core C_i does not follow this scheme as message channels are stored directly in C_i's local memory. We assume the RTOS is able to distinguish between local and global communications in order to avoid unnecessary arbitrations. It is worth noting that these assumptions on the software stack are general enough to allow our technique to be used underneath the inter-task communication mechanisms provided by specifications such as ARINC-653 and AUTOSAR.

In the following we adopt the standard TDMA naming convention. On a TDMA bus each of the n contenders is assigned a *time slot*. We use the symbol ss to indicate the

duration of a slot in clock cycles. A set of n time slots is called a *frame*. Similarly to slots, we use the symbol fs to indicate the duration of a frame in clock cycles. Inside a frame each contender is granted a dedicated slot to access the interconnect and transfer data: a *chunk* is the largest portion of memory that a task can transfer to/from global memory inside a TDMA slot when run in isolation. We call cs the size of a chunk expressed in bytes. Since a conveyed message may consist of several chunks we use ms to address the message size in bytes. Accordingly, $fb(t_i)$ refers to the start time of the current frame at time t_i (with respect to TS) and $sb(C_i, t_i)$ refers to the start time of the first available slot to core C_i after time t_i. Table 1 summarizes the notation used in the paper.

fs	Size of a frame in clock cycles with respect to TS
ss	Size of a slot in clock cycles with respect to TS
cs	Size of a chunk: amount of bytes transferred in one slot
ms	Size of a message in bytes
$fb(t_i)$	Start time of the current frame at time t_i
$sb(C_i, t_i)$	Start time of the first slot assigned to core C_i following t_i

Table 1 Notation

3.2 Software-enforced TDMA

The actual operation and efficiency of a SW-TDMA scheme depend on how *frame*, *slot* and *chunk* size, as defined in the previous section, are configured. We start from defining the size of a chunk as it affects the other two parameters. On hardware implementations of TDMA, the value cs and the size of a slot in clock cycles (ss) are part of platform's design choices along with other hardware characteristics (e.g. the bus operating frequency). Under these circumstances, the size of a slot is subject only to physical constraints and is designed so that the maximum bandwidth is achieved: the smaller possible size that permits the transmission of exactly cs bytes. On SW-TDMA, however, other sources of overhead have to be taken into account. The value ss, expressed in clock cycles with respect to the shared time source TS, must in fact consider (i) the cost of transferring cs bytes to global memory (with no contenders), and (ii) the cost of arbitrating the request. The exact value of ss can be either computed via static analysis or derived by means of extensive measurements. Once ss is known, the size of a frame fs is straightforwardly computed as $n \cdot ss$. As in standard TDMA, each core is allocated to its own slot inside a frame. We call $slot[C_i]$ the slot allocated to core C_i.

As TDMA divides time into frames, to be able to select the slot in which each core is allowed to transfer the RTOS must be able to identify, at each time t_i, the start time of the *current frame*. At every time t_i we define the current frame as the frame starting at time $fb(t_i)$, where $fb(t_i)$ is such that $t_i \geq fb(t_i)$ and $t_i < fb(t_i) + fs$. Basically $fb(t_i)$ can be computed through Equation 1 below.

$$fb(t_i) = fs \cdot (\lfloor t_i/fs \rfloor) \tag{1}$$

It is worth noting that if the size of a frame is a power of 2 ($fs = 2^{bits}$) the above operation has a blazingly fast implementation using bit-shifts: $fb(t_i) = (t_i \gg bits) \ll bits$.

Consider now an inter-core communication request issued by task τ_j running on core C_i (either a *receive_message* or *send_message*). Since a core is allowed to transfer data only inside its slot in the frame and since each slot can only hold cs bytes, the RTOS divides the data into $\lceil ms/cs \rceil$ chunks of up to cs bytes and operates each chunk transfer separately (see Figure 1). When a chunk transfer is issued on core C_i, its issuing time t_{req} is captured and the current frame's start time fb is derived using Equation 1. To correctly mimic TDMA,

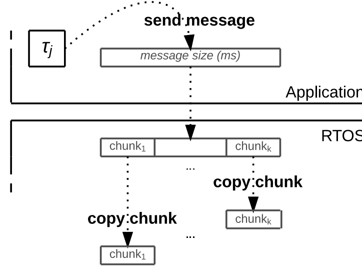
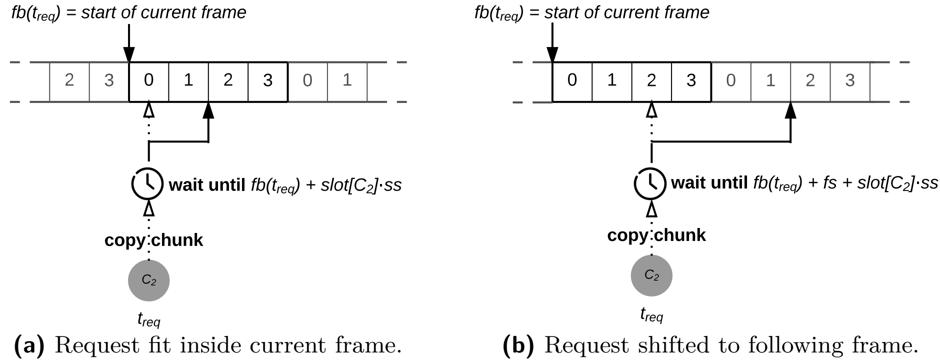


Figure 1 At the RTOS level, a message is split into $k = \lceil ms/cs \rceil - 1$ chunks of size cs and one chunk of size $ms - k \cdot cs$. Each chunk transfer will take place inside a TDMA slot assigned to C_i.

the requested transfer is deferred until the beginning of the next slot assigned to core C_i. We refer to such time as $sb(C_i, t_{req})$ and we define it as:

$$sb(C_i, t_{req}) = \begin{cases} fb(t_{req}) + slot[C_i] \cdot ss & \textbf{if } t_{req} \leq fb(t_{req}) + slot[C_i] \cdot ss \\ fb(t_{req}) + fs + slot[C_i] \cdot ss & \textbf{otherwise} \end{cases} \tag{2}$$

If the request arrived before the beginning of the corresponding core's slot inside the current frame then $sb(C_i, t_{req})$ is set to such value. Otherwise, the request missed its slot in the current frame and has therefore to wait until its slot in the next frame. The meaning of Equation 2 is graphically represented in Figure 2. Two alternative approaches are possible to intercept time $sb(C_i, t_{req})$: we may either *poll* on a cycle counter until it reaches $sb(C_i, t_{req})$, or use a *timer interrupt*. Polling is simple to implement but requires a time source to be locally available to each core (no interference in accessing it). The alarm-based solution avoids polling and works well even if no core-local time source is available but introduces additional delays due to interrupt handling.



(a) Request fit inside current frame. **(b)** Request shifted to following frame.

Figure 2 SW-TDMA arbitration examples. In Figure 2a, a request arrives before its slot and is served in the current frame. In Figure 2b, on the other hand, a request arrives after the corresponding slot in the current frame and is therefore shifted to the next one.

3.3 Bandwidth Reservation TDMA

A more flexible arbitration scheme can be defined on top of the baseline SW-TDMA arbitration, to provide higher bandwidth guarantees to a selected subset of cores. When standard TDMA is used, a frame is divided into n slots, where n is the number of contenders (cores, in our SW-TDMA): to allocate more bandwidth to some of the cores a frame is divided into m slots, with $m > n$. Once each core is associated to a slot, the remaining $m - n$ slots can be used to reserve more bandwidth to selected contenders. Let us call $core[j]$ the core that has been allocated to the j-th slot, with $j \in [0, m-1]$.

Once a chunk request is issued on core C_i, the request time t_{req} is saved and the start time of the current frame, $fb(t_{req})$, is computed following exactly Equation 1. Then, the starting time $sb(C_i, t_{req})$ of the first slot at which core C_i is allowed to transfer data must be selected similarly to the basic TDMA case, with the only difference that more than one slot may be allocated to the same core within a given frame. This difference is reported in Equation 3 where the term $first$ stands for the first slot granted to core C_i.

$$sb(C_i, t_{req}) = \begin{cases} fb(t_{req}) + first \cdot ss & \textbf{if } \exists first \mid first = \min_{j \in [0, m-1]} \left\{ \substack{core[j]=C_i \,\wedge \\ t_{req} \leq fb(t_{req}) + j \cdot ss} \right\} \\ fb(t_{req}) + fs + first \cdot ss & \textbf{otherwise } (first = \min_{j \in [0, m-1]} core[j] = C_i) \end{cases} \quad (3)$$

We fall in the first case of Equation 3 when a slot $first$ allocated to C_i exists in the current frame and its start time follows t_{req}. If there is no such slot, the second case is taken and transfer is deferred to the C_i's first allocated slot in the next frame. Figure 3 depicts the first case of Equation 3.

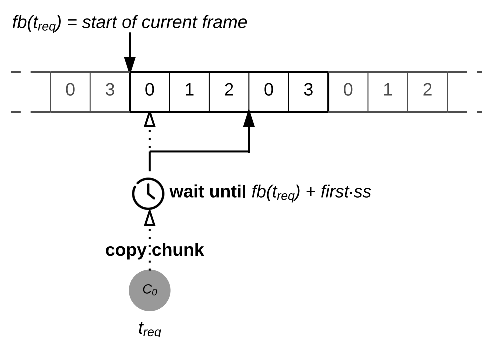


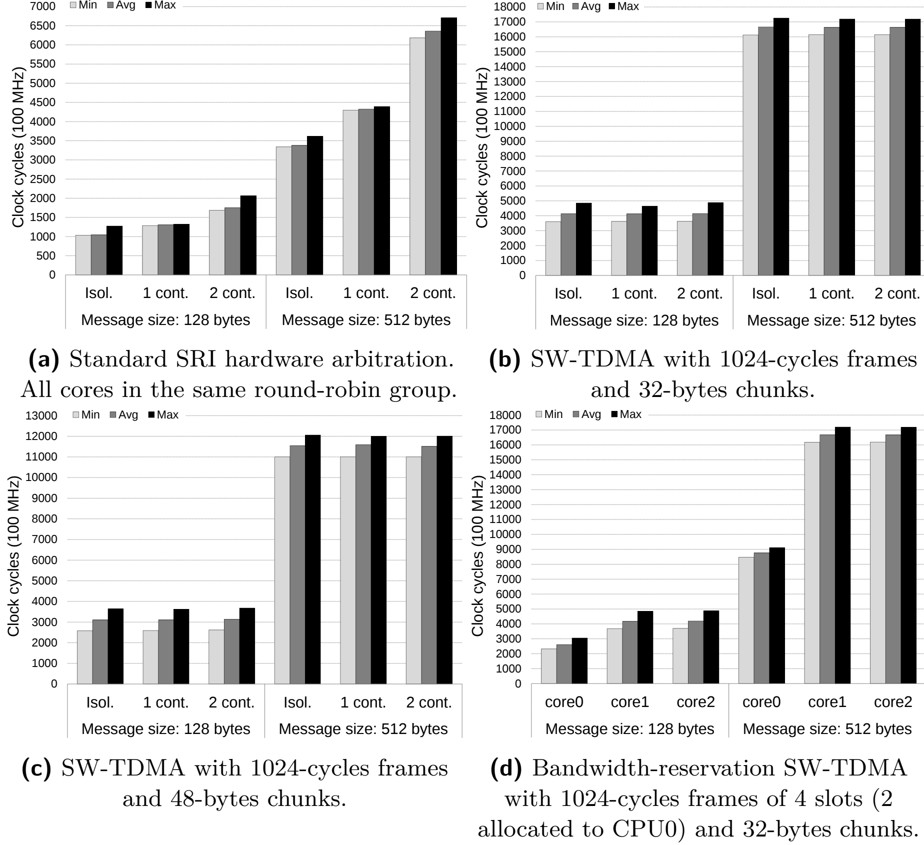
Figure 3 A chunk transfer request is assigned to the first available slot for core C_0. Slot 0 is allocated to C_0 but its starting time already passed. The next slot available is number 2 and is therefore granted. Providing multiple slots to C_0 prevents from shifting to the next frame.

It is worth noting that the bandwidth-reservation SW-TDMA scheme can also support run-time changes of the slot allocation policy. The RTOS, in fact, might decide to do so in reaction to certain events or changes in the operational mode.

4 Proof-of-Concept Evaluation

We now evaluate how well our software-based arbitration scheme is able to enforce TDMA arbitration via software on top of interconnects employing different hardware arbitrations (e.g. round-robin). We want to demonstrate that our SW-TDMA suffers no variable interference depending on the number of contenders. Finally, we want to prove that bandwidth reservation TDMA can be used to provide different bandwidth guarantees to different cores. We developed a proof of concept implementation of our approach on a realistic scenario within the automotive application domain, where silicon vendors already provide multicore solutions [11] and the AUTOSAR standard advocates partitioned multicore RTOS [3]. We run our experiments on an Infineon AURIX TC277TU [12] equipped with three TriCore CPUs. Each core has local data and instruction scratchpads. The TC277TU comes with a Shared Resource Interconnection (SRI), a crossbar that connects all cores to the Local Memory Unit (LMU) and to the Program Memory Unit (PMU). The LMU controls a shared 32KB SRAM while the PMU controls three banks of flash memory. The SRI crossbar implements both priority-driven and round robin arbitrations. Priority levels from 0 to 7 are provided. Only levels 2 and 5 can be assigned to more than one core, arbitrated according to standard round-robin

policy. In our experimental setup all cores are in the same round-robin group. In addition, the crossbar employs separate hardware channels for each slave resource connected to it. That is, different SW-TDMA arbitrations can be carried out independently for each of the crossbar's slaves.



(a) Standard SRI hardware arbitration. All cores in the same round-robin group.

(b) SW-TDMA with 1024-cycles frames and 32-bytes chunks.

(c) SW-TDMA with 1024-cycles frames and 48-bytes chunks.

(d) Bandwidth-reservation SW-TDMA with 1024-cycles frames of 4 slots (2 allocated to CPU0) and 32-bytes chunks.

Figure 4 Execution times in clock cycles (100MHz) for message send primitives with messages of size 128 and 512 bytes. Figures 4a, 4b and 4c report the cost for sending a message in isolation or with one or two contenders respectively; min and max are the absolute values observed on any of the cores. In Figure 4d all cores run a task sending messages, min and max values are per-core.

Our SW-TDMA arbitration has been implemented in ERIKA Enterprise [7], an OSEK/VDX compliant RTOS. Test applications are developed on top of ERIKA and make use of a user-level library implementing the AUTOSAR IOC communication layer. Tasks' data and instructions are placed on cores scratchpads while IOC buffers are located inside the shared SRAM. Test applications are designed to measure the cost of sending messages of different sizes under different workloads: a single task runs on each core and all three tasks send messages of the same size to different buffers (no mutual exclusion across tasks is required). Tasks share the same period and are synchronized through a barrier so that to produce maximum interference. Measurements are collected via the on-core cycle counter running at 100MHz and are stored in the scratchpad (low latency tracing with no contention). As a first experiment, we evaluated the cost of accessing the SRAM through the crossbar *as-is*, with no modification. Results observed for sending messages of 128 and 512 bytes are reported in Figure 4a. The cost of sending a message evidently grows with the number of contenders. We witnessed an increase of 27% and 22% in the maximum observed execution time (MOET) when two tasks try sending a message (of 128 and 512 bytes respectively) at the same time, with respect to performing the same activity in isolation. Adding one more

contender causes the MOET to become 93% and 86% bigger than in isolation, for messages of 128 and 512 bytes. This non-linear growth of interference can be explained by the very structure of the SRI. The AURIX crossbar, in fact, divides a transaction into two phases: i) a *request phase* where the address is sent and the request is arbitrated, and ii) a *data phase* where actual data is transmitted. No more than one request can be arbitrated at a time, however, data and request phases of different transactions can be pipelined. This also explains why the MOET is only $\sim 90\%$ higher when all the cores access the crossbar. Different access patters, however, may cause more interference to occur.

In Figure 4b we report the execution times observed when sending messages of 128 and 512 bytes under SW-TDMA, with frames of 1024 clock cycles and chunks of 32 bytes. The graph shows that the cost for sending a message does not depend any more on the number of contenders, which is exactly what we were after. It is worth understanding which portion of a slot is wasted to take software arbitration decisions. Under the above configuration a slot is approximately 342 clock cycles. By means of extensive measurements, we observed that a core in isolation is capable of transferring up to 57 bytes in 342 clock cycles. Therefore, if software arbitration with chunks of 32 bytes is employed, the throughput of a core is reduced by a 44%. In Figure 4c, however, we show that throughput can be enhanced while maintaining isolation across cores. The graph reports a different configuration in which frames are still of 1024 clock cycles but chunks are now of 48 bytes. The cost of sending a message is still constant irrespective of the number of contenders while the throughput is increased and execution times decreased. Under this configuration SW-TDMA reduces the throughput of a core in a slot only by a 16%. As expected, when comparing our approach to the original SRI crossbar we notice that the average cost of sending a message is increased by almost 80%, for messages of 128 bytes. The magnitude of this result is explained by the fact that the AURIX SRI crossbar also allows some degree of pipelining. A fair comparison would require to evaluate our SW-TDMA against pure TDMA or RR interconnects with no splitting. In particular, the results obtained for SW-TDMA are extremely near to the theoretical performance offered by hardware TDMA. From the mixed-criticality standpoint, SW-TDMA caters for isolation among criticality levels, regardless of bus access patterns.

The results we observed on bandwidth-reservation TDMA are shown in Figure 4d. The frame is again set to 1024 cycles but divided into 4 slots for chunks of 32 bytes. Slot 0 and slot 2 are both allocated to core 0. Results show that, as expected, sending a message on core 1 and core 2 costs exactly as in Figure 4b where a frame of 1024 was divided into 3 slots each for a chunk of 32 bytes. Core 0 instead executes twice as fast, as it is granted two slots. Our evaluation highlights that bandwidth-reservation SW-TDMA allows assigning larger portions of the shared interconnect to a selection of cores, in keeping with mixed-criticality requirements. Bandwidth reservation TDMA enables more complex arbitrations, while still preserving isolation.

5 Conclusion

In this paper we propose a solution to enforce isolation among cores accessing shared resources that does not rely on the availability of specific hardware. SW-TDMA is particularly interesting in the context of mixed-criticality applications where critical tasks must not be interfered by lower criticality tasks. We provided experimental evidence that SW-TDMA ensures isolation while introducing only a 16% throughput drop due to software arbitration. A more flexible scheme, named bandwidth reservation TDMA, has been also introduced and evaluated. The latter technique provides selected cores with more bandwidth without

affecting isolation and is much suited to guarantee a better quality of service to critical tasks. As future work we plan to investigate more complex schemes as, for instance, random permutation. We also plan to further reduce the overhead introduced by software arbitration.

Acknowledgment. The work presented in this paper has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 611085 (PROXIMA, www.proxima-project.eu).

References

1 ISO 17458-1. Road vehicles – FlexRay Communications System – Part 1, 2013.
2 APEX Working Group. Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface, 2003.
3 AUTOSAR. AUTOSAR Release 4.1. http://www.autosar.org/, 2014.
4 A. Burns and R. Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2015.
5 D. Dasari et al. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *Proc. of IEEE 10th Conference on Trust, Security and Privacy in Computing and Communications*, 2011.
6 R.I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 2011.
7 Evidence. ERIKA Enterprise. http://erika.tuxfamily.org/, 2015.
8 G. Giannopoulou et al. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *Proc. of ACM International Conference on Embedded Software*, 2012.
9 G. Giannopoulou et al. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proc. of 11th ACM Intl. Conference on Embedded Software*, 2013.
10 D. Goswami et al. Time-triggered implementations of mixed-criticality automotive software. In *Proc. of 13th Design, Automation & Test in Europe Conference*, 2012.
11 Infineon Technologies AG. AURIX™ TriCore™. http://www.infineon.com/aurix, 2012.
12 Infineon Technologies AG. TC27x Manual. http://www.infineon.com/aurix, 2014.
13 International Electrotechnical Comission. *IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Edition 2.0*, 2009.
14 T. Kelter et al. Evaluation of resource arbitration methods for multi-core real-time systems. In *13th International Workshop on Worst-Case Execution Time Analysis*, 2013.
15 R. Pellizzoni et al. Worst case delay analysis for memory interference in multicore systems. In *Proc. of 13th Conference on Design, Automation and Test in Europe*, 2010.
16 J. Rosen et al. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of 28th IEEE Real-Time Systems Symposium*, 2007.
17 A. Schranzhofer et al. Timing analysis for resource access interference on adaptive resource arbiters. In *Real-Time and Embedded Technology and Applications Symposium*, 2011.
18 H. Shah et al. Priority division: A high-speed shared-memory bus arbitration with bounded latency. In *Proc. of Design, Automation & Test in Europe Conference*, 2011.
19 L. Sigrist et al. Mixed-criticality runtime mechanisms and evaluation on multicores. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015.
20 S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. of 28th IEEE Real-Time Systems Symposium*, 2007.
21 H. Yun et al. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Proc. of 24th Euromicro Conference on Real-Time Systems ECRTS*, 2012.