Downloaded from orbit.dtu.dk on: Jan 23, 2021

DTU Library

Time-predictable synchronization support with a shared scratchpad memory

Maroun, Emad Jacob; Hansen, Henrik Enggaard; Kristensen, Andreas Toftegaard; Schoeberl, Martin

Published in:
Microprocessors and Microsystems

Link to article, DOI:
10.1016/j.micpro.2018.09.014

Publication date:
2019

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
Maroun, E. J., Hansen, H. E., Kristensen, A. T., & Schoeberl, M. (2019). Time-predictable synchronization support with a shared scratchpad memory. *Microprocessors and Microsystems*, *64*, 34-42. https://doi.org/10.1016/j.micpro.2018.09.014

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Time-predictable Synchronization Support with a Shared Scratchpad Memory

Emad Jacob Maroun, Henrik Enggaard Hansen, Andreas Toftegaard Kristensen, Martin Schoeberl*

Department of Applied Mathematics and Computer Science
Technical University of Denmark
Kongens Lyngby, Denmark

Abstract

Multicore processors need to communicate when working on shared tasks. In classical systems, this is performed via shared objects protected by locks, which are implemented with atomic operations on the main memory. However, access to shared main memory is already a bottleneck for multicore processors. Furthermore, the access time to a shared memory is often hard to predict and therefore problematic for real-time systems.

This paper presents a shared on-chip memory that is used for communication and supports atomic operations to implement locks. Access to the shared memory is arbitrated with time division multiplexing, providing time-predictable access. The shared memory supports extended time slots so that a processor can execute more than one memory operation atomically. This allows for the implementation of locking and other synchronization primitives.

We evaluate this shared scratchpad memory with synchronization support on a 9-core version of the T-CREST multicore platform. Worst-case access latency to the shared scratchpad is 13 clock cycles. Access to the atomic section under full contention, when every processor core wants access to acquire a lock, is 135 clock cycles.

1. Introduction

Multicore processors are the current standard solution to increase the performance of applications. Several threads, executing on several cores in parallel, provide a speedup compared to a single core processor. However, when an application is organized into several parallel running threads, those threads working towards a common goal need to communicate. The standard way of multithread communication is via objects allocated in shared memory with access protected by locks. To allow the processors to cooperate on the same application, these systems require the support of synchronization mechanisms for concurrent access to shared data or external devices. When such embedded systems have timing requirements, such as real-time systems, the access to shared data and the synchronization mechanism must be predictable in order to allow for analyzable worst-case execution time (WCET).

All synchronization primitives require some hardware support in order to guarantee atomicity. This is typically done using hardware supported atomic instructions, which are supported by the memory controller. In standard processors, the shared memory is external DRAM memory and the synchronization support, such as compare and swap, is implemented on top of this external memory.

This paper presents a shared scratchpad memory (SSPM) with hardware support for synchronization and interprocessor communication. We avoid costly operations on external memory by using on-chip memory. The memory access is arbitrated in a time-division multiplexing (TDM) scheme where each core receives a one cycle access slot each TDM round. This enables WCET analysis of memory accesses and therefore of applications. The SSPM allows the extension of an access slot for multiple cycles to allow more than one memory access. This extended access slot enables atomic operations on the SSPM to provide efficient and time-predictable synchronization primitives. The proposed solution is integrated into the hard real-time T-CREST multicore platform [18]. The implementation is a peripheral hardware connected to the processing cores and is using the OCP bus protocol for the arbitration.

As we aim for a time-predictable system to support real-time systems, we analyze the WCET of memory accesses and the usage of the extended slot for synchronization. On a 9-core version of the T-CREST multicore processor, the worst-case access latency to the shared scratchpad is 13 clock cycles. Worst-case access latency to the atomic section under full contention, when every processor core wants access to acquire a lock, is 135 clock cycles. Furthermore, we present the average-case execution times for interprocessor communication, using both the SSPM and the Argo NoC [10] of the T-CREST platform implemented on the Altera Cyclone EP4CE115 device.

This paper is an extension of [5], where we presented the first version of the SSPM and provided an initial evaluation of the SSPM. We extend this work with following additional

*Corresponding author
Email address: masca@dtu.dk (Martin Schoeberl)

contributions: (1) more in-depth description of the SSPM, (2) extending the evaluation section with multichannel tests, (3) a complete integration with the main T-CREST project, and (4) providing complete instructions how to build the SSPM and how to reproduce the results.

This paper is organized into six sections: Section 2 presents work related to hardware mechanisms for synchronization support and interprocessor communication. Section 3 provides background on the T-CREST platform. Section 4 presents the design and implementation of the SSPM with synchronization support. Section 5 presents the results of the implemented solution and evaluates it against the Argo NoC. Section 6 concludes the paper.

2. Related Work

The use of hardware for lock-based synchronization mechanisms is explored in [23]. The authors present best- and worst-case results for the synchronization primitives implemented. They implement a global locking scheme to make access to a shared memory atomic and they also present support for barrier synchronization. They provide memory access results using two cores. For single access performance using their coherence mechanism, it takes 4 cycles best-case and 7 cycles worst-case. For our solution, it only takes 6 cycles worst-case to gain access to the memory when it is shared between two cores.

In IA-32, bus locking through the LOCK instruction guarantees the atomic execution of other instructions, e.g. compare-and-swap [8]. Bus locking reserves the memory bus for subsequent instructions allowing for the compare-and-swap instruction to complete in a globally consistent manner. With our extended time slot, we support atomicity for arbitrary memory operations. Therefore, an atomic compare-and-swap operation can be implemented on top of the extended time slot.

Many soft-core processors, such as the Xilinx MicroBlaze and the Patmos processor, do not support atomic instructions for synchronization. The authors in [24] present hardware implementations of basic synchronization mechanisms to support locking and barriers for the soft-core Xilinx MicroBlaze processor. For six processors, their presented approaches take around 43-46 cycles to execute the lock primitive, while our solution has an average execution time of 45 using multiple slots and 106 using single slots, as discussed in Section 5.5. Their solutions require the use of specific ports on the Xilinx MicroBlaze, while our shared scratchpad can be used with any unmodified core to support atomic operations, given that it supports memory-mapped peripherals.

An unorthodox alternative to support barrier synchronization is presented in [17]. Threads are forced to wait by invalidating instruction caches lines that contain the execution point at which the threads should synchronize. Additional hardware for the memory subsystem is implemented in order to filter requests to these cache lines and refuse to serve them until a specific condition is met. Our implementation is likewise exploiting stalling to enforce synchronous execution, but our local storage doubles as an atomic data sharing mechanism.

In [9] an implementation of a message passing interface for the Tile64 processor platform from Tilera is presented. The data cache is used for loading messages, resulting in high cache miss costs for large messages. The usage of the caches also complicates the timing analysis for WCET analysis. The method presented in this paper avoids this by using the SSPM for communication.

Regarding message passing in a time-predictable network-on-chip (NoC), the authors of [22] evaluate the Argo NoC of the T-CREST platform for message passing and analyze its WCET, providing the end-to-end latency for core-to-core messages. Message passing is also used to implement locking and barriers. As we build the SSPM for the same platform, we complement that solution. However, a shared memory solution does not scale as well as a NoC for a greater number of processors [2, 12]. We confirm this for some of the benchmarks presented in Section 5.3, so a choice between the SSPM and NoC solution will be application specific.

In [15] a distributed shared memory architecture is explored based on a NoC and a shared portion of the addressing space to support synchronization and data exchange between processors. The authors use a number of benchmarks which are either computation intensive (FFT and vector normalization), memory intensive (matrix multiplication) or synchronization intensive (matrix factorization and sort) and consider the effect of having only a single shared memory and distributing the shared memory modules. For synchronization and memory heavy tasks, there is a reduction in execution time and energy when using distributed shared memory, while the computation intensive tasks only see a minor improvement. We only consider the use of a single shared memory, so in terms of energy and execution time, this design is more appropriate for computation intensive tasks.

It is argued that multiprocessors should support both message-passing and shared-memory mechanisms, since one may be better than the other for certain types of tasks [11]. Message-passing architectures are found to be suitable for applications where the data packets are large, or the transfer latency is insignificant compared to other computations. Shared-memory architectures prevail in situations with appropriate use of prefetching. In their comparison, the performance is equivalent. They present an architecture supporting both mechanisms for interprocessor communication.

3. The T-CREST Platform

This section provides an overview of the T-CREST multicore platform that we used to implement and evaluate our scratchpad memory with synchronization support. T-CREST is a processor designed especially for hard-real time systems with the goal of having a low WCET and

to ease its analysis [18]. All components have thus been designed with a focus on time-predictability and to reduce the complexity and pessimism of the WCET analysis. The platform consists of a number of processing nodes and two networks-on-chip (NoCs): A NoC for message passing between cores called Argo [10] and a shared memory access NoC. Both NoCs use TDM for arbitration in the same way as we use it for the SSPM.

A processing node consists of a RISC processor called Patmos [20], special cache memories, such as the method cache, and a local scratchpad memory. The method cache is an important feature for the support of the atomic instructions in the SSPM. It caches full methods. Therefore, the cache can only miss at a call or return instruction. All other instructions are guaranteed hits. That feature supports uninterrupted executed instructions for the atomic region.

Patmos is supported by a compiler developed with a focus on WCET [16], based on the LLVM framework [14]. The compiler supports two static WCET analysis tools: the aiT [6] tool from AbsInt and the open-source tool platin [7]. Both tools allow static computation of WCET bounds and support the specific architecture of Patmos.

The Argo NoC [10] provides message passing to support inter-processor communication and offers the possibility to set up virtual point-to-point channels between processor cores. Data can be pushed across these circuits using direct memory access controllers in the source end of the circuit, transmitting blocks of data from the local SPM into the SPM of the remote processor.

In order to communicate between two processors, the Argo NoC uses a static time-division TDM schedule for routing packets through communication channels in routers and on the links. The network interface between the NoC and the processor node integrates the direct memory access controllers with the TDM scheduling such that no flow control or buffering is needed.

Different types of data can thus be transferred on the NoC, e.g. message passing data between cores and synchronization operations. The Argo NoC can thus be used to support synchronization primitives [22]. Being a distributed system, the Argo NoC can be used to implement any distributed locking algorithm directly. Apart from the NoC, it is possible to implement software-based methods for synchronization in T-CREST. The data cache can be invalidated [19] and Lamport's bakery algorithm [13] can be used to implement locks. This software implementation has a very high overhead and is therefore impractical. Our presented SSPM provides a better and more efficient implementation for locks.

4. Design and Implementation

The SSPM consists of an on-chip scratchpad memory, an arbitration circuit in front of this memory, and an interface for each processor core. Figure 1 sketches this setup. From the perspective of a single processor core the SSPM behaves as a conventional scratchpad on a peripheral device bus with standard read and write commands. The only difference is the timing of those read or write commands.

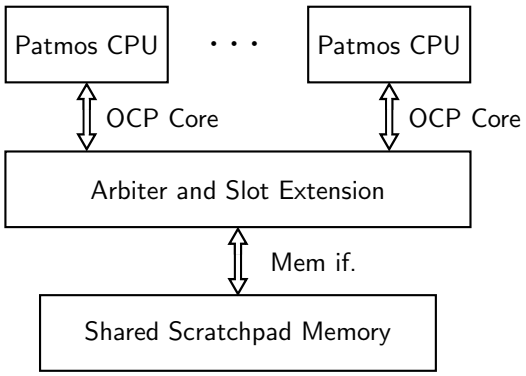


Figure 1: The SSPM connected via the arbiter to the processor cores.

4.1. Time-predictable Shared On-Chip Memory

The core interfaces connect each core to the arbitration circuit, keep read and write requests (referred to as commands) in a buffer, and handle the protocol with a core associated with each cores interface. The arbiter multiplexes access to the memory using TDM with time slots, ensuring both evenly distributed access for all cores and arbitration of concurrent access. Only within its time slot can a respective core have its read or write commands executed. TDM based arbitration is the basis for time-predictable shared memory access [21].

If a core requests access outside of its time slot, the core interface will stall the core by not issuing a command response. The bus protocol dictates that a core must not issue new commands before a response of an outstanding command has been received. By withholding this response the SSPM can control the flow of commands from cores. Whilst cores are stalled, the command is buffered, but any reads or writes are not executed. When the time slot arrives, the arbiter allows for the fulfillment of the command by executing the buffered command and issuing the response, allowing the core to resume execution. Consequently, the use of time slots incurs a minimum interval of $n - 1$ cycles between subsequent commands from a single core, where n is the number of cores with access to the SSPM. Time slots will also ensure that simultaneously issued commands will execute in a deterministic order, albeit the first to access is dependent on the currently active time slot.

4.2. Atomicity Through Bus Locking

Support for atomic operations, such as acquiring a lock, is implemented by granting exclusive access to the SSPM to a single core for multiple clock cycles. This allows for the fulfillment of multiple reads and writes on the SSPM without interference from other cores, thus implementing the global behavior of atomic operations.

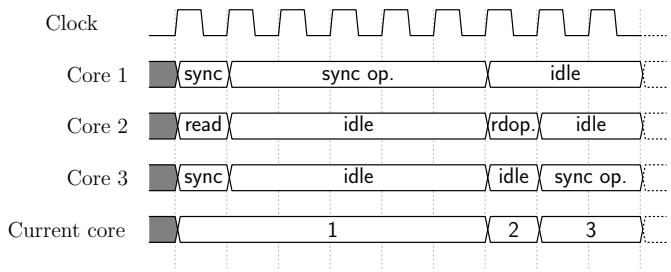


Figure 2: An example showing the arbiter behavior when multiple cores request an action simultaneously. When cores are waiting, they go idle until their time slot arrives.

A core can request such an extended time slot from the SSPM by issuing a read request from a special address. When the core's time slot arrives, the arbiter grants an extended time slot to allow for fulfilling multiple commands consecutively. Those multiple commands, which are executed atomic, are used to implement a synchronization primitive, e.g., a lock.

Figure 2 shows an example of cores competing for access to the SSPM. Core 1 and core 3 request extended time slots (`sync`) and core 2 has an outstanding `read` command. The current time slot ("Current core") is for core 1 and the extended time slot is granted in the following cycle. In the meantime, core 2 and core 3 are stalled and go idle. When the extended time slot has passed, the arbiter grants access to core 2, followed by core 3's extended time slot.

The SSPM includes a buffer to meet timing requirements and commands are thus always delayed by one cycle. It is possible to implement a lock in 3 instructions, hence a minimum of 6 cycles is needed in the extended time slots to guarantee atomicity for such a locking mechanism.

The extended time slots will influence the worst-case and average-case performance of loads and stores in the SSPM. In a situation where no cores are using extended time slots, the observed latency is only influenced by the alignment with regular time slots.

In a worst-case scenario, every core requests an extended time slot. In this case, the worst-case delay for a command, be it read, write or request of an extended time slot is $(n-1)c_{\text{ets}}$. n is the number of cores and c_{ets} is the number of cycles an extended time slot uses.

4.3. Single Extended Time Slot

In the above implementation (the *multi slot* implementation), the potentially large delays from extended time slots could unnecessarily impede performance. It is assumed that programs requesting extended time slots can tolerate longer delays, whereas the delays for conventional reads and writes should be shorter. These two assumptions allow for an alternate design of the arbiter.

When read or write commands are handled, the SSPM behaves as the previous case with the multi slot arbiter: a single time slot per core, taken in sequence and with no priority. For the *single slot arbiter*, the only difference is that if one core requests an extended time slot and has it granted, then another core cannot be granted an extended time slot before a whole TDM "round" has passed. By a "round" it is meant that every core must get a chance to fulfill a normal read and write to the SSPM before a new extended time slot is granted. Therefore, the delay by the extended time slot of normal read and write commands is reduced.

A flag is used for tracking if an extended time slot has been granted in the last round and which core it was assigned to. As long as this flag is set, no core can be granted an extended time slot. This flag is reset when the respective core is granted a conventional time slot. This ensures that one core can not immediately be granted an extended time slot again, but instead the following core is the next candidate for receiving it. Note that the slot is still dynamically present and takes its place from the conventional time slot of the respective core.

By limiting the arbiter to grant one extended time slot per round, a worst-case delay between commands becomes: $n - 2 + c_{\text{ets}}$, where n is the number of cores. The two subtracted cycles account for the core with an extended time slot and the core making the read or write command. For requesting an extended time slot the worst-case delay becomes: $n \times (n + c_{\text{ets}})$. The breakdown is: $n \times c_{\text{ets}}$ for the delay from extended time slots and n^2 for the delay from only having one extended time slot per round.

4.4. Hardware Implementation

We use the T-CREST multicore processor for our implementation. The Patmos processor is written in the hardware construction language Chisel [1]. Therefore, we implemented the SSPM in Chisel as well.

The SSPM consists of a memory component, the arbitration, and slot extension component, and interface components to the processor cores. Each component also contains test-benches for individual component tests.

For the integration into the T-CREST multicore, we had to extend the multicore generation scripts. In the near future, this step will not be needed, as we plan to provide a multicore top-level in Chisel.

4.5. Lock Implementation

The length of the extended time slot can be configured and we can perform arbitrary operations within this extended time slot, which are atomic in relation to the SSPM. For example, we can support compare-and-set for single words or multiword compare-and-set. As an example we show a simple test-and-set based locking mechanism using 3 instructions:

```
load r0, &sync_request_address
load r1, &lock
store &lock, 1
```

The extended time slot is requested with the first load from the request address. When the extended time slot

Table 1: Resource consumption of a 9-core version of T-CREST with the SSPM.

Entity	LUTs	Registers	RAM bits
Patmos cores	90 243	43 877	1 344 192
Argo NoC	15 077	8 342	99 072
Main memory arbiter	1828	765	0
Main memory controller	451	331	0
SSPM: multi slot arbiter	615	462	-
SSPM: single slot arbiter	635	467	-

arrives, the second load and the following store can be served uninterrupted. When the load of `&lock` returned a 1 (locked), then the sequence of instructions changed nothing, and the processor did not acquire the lock. When the load of `&lock` returned a 0, the lock was free and the setting to 1 acquires the lock.

The access to the extended time slot is only served at the beginning of an extended slot. Therefore, the processor shall not stall for any reason or be interrupted between the request of the extended time slot and the execution of the atomic operations. If interrupts are used, they need to be turned off before this atomic section. One issue could be a cache miss between the request instruction and the use instructions. However, with Patmos and the method cache, the cache misses happen only at function call or return. All other instruction fetches are guaranteed hits. For a processor with a normal instruction cache, a solution against the cache miss is to align the request and atomic instructions within a cache line.

5. Evaluation and Results

This section presents the evaluation of the SSPM within the T-CREST multicore processor. We use Altera Quartus (v16.1), targeting the Altera Cyclone FPGA (model EP4CE115) and set Quartus' optimization mode to "Performance (Aggressive)". The clock frequency was set to 80 MHz, as this is the default setting for the Patmos core in this FPGA. We implemented a 9-core design by adding the SSPM to the default implementation of T-CREST with the Argo NoC. The Argo NoC is configured with the default all-to-all schedule and message length of one header word and two data words. It has to be noted that the Argo NoC can also be configured with application specific schedules and different message length per channel.

We tested two variations of the SSPM: (1) using the single slot and (2) the multi slot version. The evaluations against the Argo NoC were performed using the single slot implementation since none of the tests for comparison use extended time slots, which is the only difference between the two implementations.

5.1. Resource Consumption

Table 1 presents the resource consumption of the Patmos cores, the Argo NoC, the main memory arbiter, the main memory controller, and our SSPM in lookup tables (LUTs), registers, and RAM bits. We do not show the size of the scratchpad memory itself, as it is configurable. Relative to the Patmos processors and the Argo NoC the hardware consumption of the SSPM is low. Even compared to the memory arbiter and controller for main memory, our solution is also relatively cheap. The two arbitration schemes have similar resource consumption; the single slot implementation is expected to have a slightly larger utilization due to the flag checks.

Table 2: Worst case latency (WCL) for access to the SSPM in clock cycles (r/w) for a 32-bit word. ETS shows the delay from requesting an extended time slot until it is provided.

	Single slot		Multi slot
Cores	WCL (r/w)	WCL (ETS)	WCL (r/w/ETS)
2	6	16	6
4	8	40	18
9	13	135	48
16	20	352	90
32	36	1216	186
64	68	4480	378

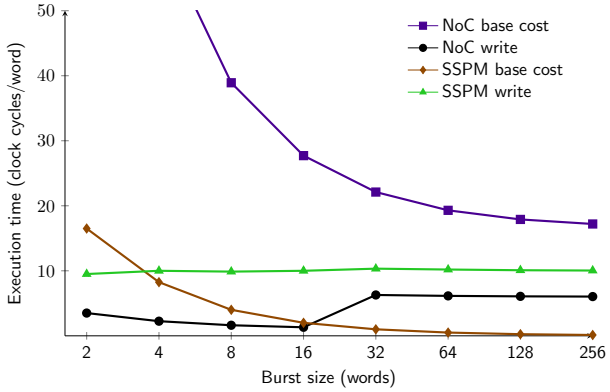
5.2. Worst-Case Latency

Table 2 shows the worst-case latencies of read or write (r/w) operations for 32-bit words and the extended time slot (ETS) allocation for the single slot and the multi slot version of the SSPM. The single slot version gives the lowest worst-case latency for reads and writes. It is basically the length of the TDM round plus some extra cycles for pipelining. The multi slot version increases the worst-case memory access time, but provides a shorter latency when requesting an extended time slot for atomic operations. As the single slot version only allows a single slot extension per TDM round, the worst case of granting an extended slot is to wait n TDM rounds for an n core system. For example, for a multicore system with 32 cores more than a thousand cycles could potentially be spent on waiting for an extended time slot.

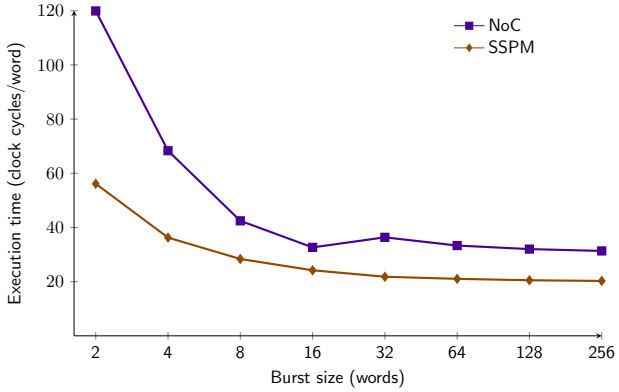
Comparing against the Argo NoC, a nine core NoC was evaluated in [22] with a worst-case latency of 211 clock cycles for a blocking transfer of two 32-bit words between two cores. With the SSPM and the multi-slot arbiter such an access needs 2×48 clock cycles for the write operation and another 2×48 clock cycles for a read operation, summing up to 192 clock cycles. With the single-slot arbiter, the same write and read operation can execute in 52 clock cycles.

The presented worst-case latencies are statically analyzed numbers. Therefore, we can use these latencies in static worst-case execution time analysis of tasks.

Comparing these results with the conclusion from [11] reveals that the SSPM is positioned as a trade-off between shared-memory and message passing. Transfer latency is



(a) Interprocessor communication execution times and scratchpad access time per word for the SSPM compared to the Argo NoC. Base cost is the cost per for sending the message until acknowledging that message.



(b) Interprocessor communication execution times per word for the SSPM compared to the Argo NoC. The scenario contains: a write, send, read, and acknowledge.

Figure 3: Interprocessor communication measurements for the Argo NoC and the SSPM.

smaller than for message-passing, but there is no need for prefetching. However, the access latency to the SSPM grows linearly with the number of cores in the system. I.e. the more cores there are in the system, the longer each core could potentially wait before getting its turn in the schedule. This is in contrast to a message-passing NoC, where the number of links increases with the number of cores and therefore also the total bandwidth of the system increases. This is supported by [3] for the general case. That means that the NoC communication is better scalable than a shared on-chip memory. Therefore, we envision to use the SSPM in a clustered organization where just a few cores share a SSPM and these clusters are connected to each other with a NoC. A reasonable cluster size might be in the range of 8 to 16 cores. The Kalray multicore processor [4] uses clusters of 16 cores that share one on-chip memory. The clusters are connected via a NoC.

5.3. Communication Performance

We measure access time and roundtrip times of messages. As reading and writing times are equal, we present writing times only. We define the roundtrip time as the time between sending the message and receiving the acknowledge from the receiver. We vary as parameter the message size.

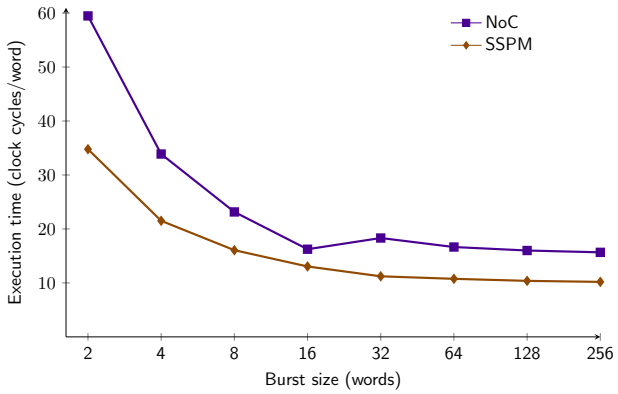
Figure 3a shows the access times of writes to the SSPM and the NoC local scratchpad memory and roundtrip time. The figure shows the time in cycles per word. We measure with increasing burst sizes executed in a tight loop. Therefore, it is expected that larger burst sizes perform better as overheads are amortized.

The write time to the SSPM is relatively constant at around 10 clock cycles, independent of the burst size. This is consistent with the TDM based memory arbitration for 9 cores. However, we can observe that the looping overhead is completely hidden by the waiting time for the TDM slot.

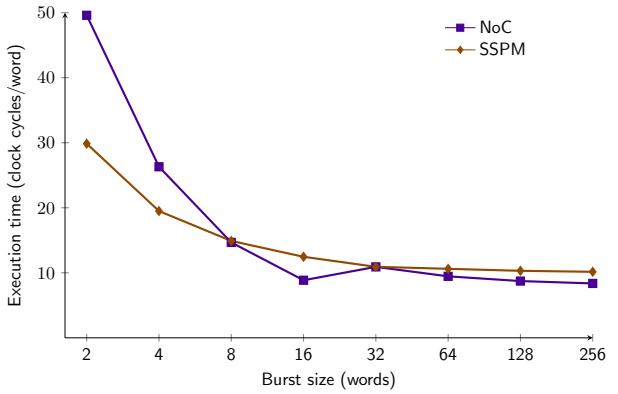
The write timing into the NoC scratchpad is shorter than the write timing into the SSPM. This is expected, as the NoC scratchpad is a local memory with one clock cycle access time. However, we observe an interesting pattern: From 2 up to 16 words the access time per word decreases, as expected. It decreases down to 1.3 clock cycles for the 16 word burst. For burst sizes of 32 and more words, the access time increases to a value of around 6 clock cycles per word. The result can be explained by loop unrolling performed by the compiler for shorter loops. For larger burst sizes the loop overhead dominates the write times.

Figure 3a also shows the time of a roundtrip using the Argo NoC and the SSPM. The figure shows the advantage of a shared memory structure compared to a message passing NoC. The Argo NoC has a high fixed roundtrip overhead, which is amortized with longer messages. Argo's fixed overhead for setting up a message to be passed over the channel is in the range of 200 clock cycles. Therefore, with two word bursts, the average per word cost is around 106 cycles, falling to 62 cycles with four word bursts. In contrast, the SSPM's overhead is a fixed 34 cycles when no cores are using extended time slots, regardless of the burst size. We expect that the setup code for the Argo NoC messages can be optimized if shorter roundtrip times are needed.

Figure 3b shows the measurements of a complete end-to-end example: A core in a computing pipeline works on some data and moves it to the next core in the pipeline. We model in our synthetic benchmark the cost of the core communication, which consists of a local data move, the data transfer (or notification in the SSPM case) to the receiver core, local data movement in the receiving core, and acknowledgment to the sender. The data movement is between a local working scratchpad and either the SSPM or the Argo NoC scratchpad. We ignore what work a core would do to the data since we are only interested in the cost



(a) Interprocessor communication execution times per word for the SSPM compared to the Argo NoC, where one core is simultaneously sending to two receiving cores. Synthetic scenario with a write, send, read, and acknowledge.



(b) Interprocessor communication execution times per word for the SSPM compared to the Argo NoC, where one core is simultaneously sending to four receiving cores. Synthetic scenario with a write, send, read, and acknowledge.

Figure 4: Interprocessor communication measurements with multiple channels per core.

of communication. In our test, the cores do no work. The figure shows the cost per word for the Argo NoC versus our SSPM. We can see that both solutions scale with larger burst sizes up to a limit where pure communication cost per word dominates the measurement. With the SSPM we reach the limit at around 20 clock cycles, which results from the cost of one write and one read access to the SSPM. The Argo approach a limit of 32 clock cycles per word for the default all-to-all schedule and a single header word overhead per 2 words of data. The results show that the SSPM is slightly more efficient at message passing than the Argo NoC when only one channel of communication is active for each core.

However, the SSPM hardware implementation does not scale as well as the Argo NoC. With an increase in core count the SSPM will struggle to meet timing constraints in the synthesis. However, the results show that it can afford to have a few more pipeline stages and still be faster than the Argo NoC.

5.4. Multiple Channels

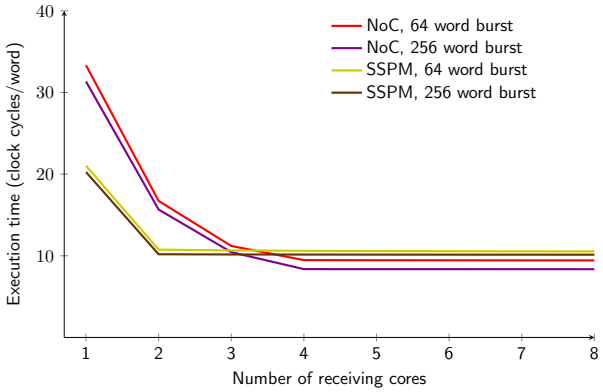
The SSPM needs to share its bandwidth between all active channels, regardless of their number and layout.

On the other hand, the Argo NoC supports a fixed bandwidth between all core pairs. Therefore, when communication channels are from one to many cores or many to one core, the NoC has the same bandwidth per channel as it has in a one to one situation, making the total bandwidth higher. The local scratchpad is not a limiting factor in these cases, as its access time is too low to affect the NoC's bandwidth.

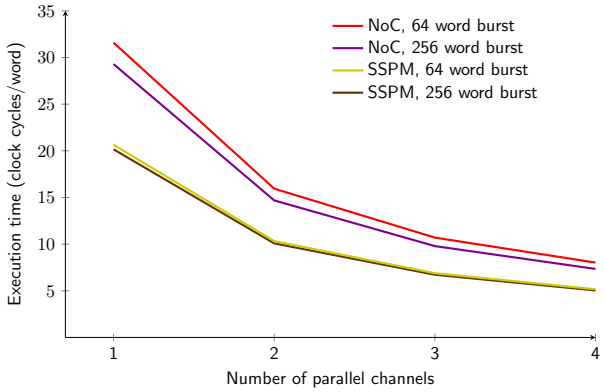
Figure 4a and Figure 4b show the same end-to-end example as before, but with multiple channels. The results are clock cycle per word. Figure 4a shows one core is sending to two receiving cores. We see that for both, our solution and when using the NoC, the amortized cost per word is lower than the single channel shown in Figure 3b. The SSPM is still faster than the NoC for all burst sizes and reaches double the bandwidth as in the single channel case. In Figure 4b, however, we see the cost per word when a core is sending to four receiving cores simultaneously. For burst sizes of 8 or higher, the NoC becomes as fast as the SSPM or up to 40% faster when the burst size is 16 words. The same holds when simultaneously sending to more than four cores.

It is worth noting that the SSPM's bandwidth is completely utilized in these benchmarks, being a maximum of one word per 10 clock cycles per core. Figure 5a shows the cost of sending a word in 64 and 256 word bursts when sending to between one and eight cores. For the SSPM, sending to just one core requires about 20 cycles; 10 for the sender to write, and 10 for the receiver to read. Sending to two cores drops the cost to half. After the sender spends 10 cycles sending to the first receiver, the first receiver will read the message while the sender is sending to the second receiver. Therefore, when the sender starts over, the first receiver would have already acknowledged the previous message. Sending to more than two cores does not increase efficiency, as the sending core cannot write to the SSPM faster than one word per 10 cycles. This is an inherent limit to the bandwidth of one-to-many communication using the SSPM.

This is not the case for the Argo NoC; the bottleneck when sending to multiple cores is the sending core itself. If the NoC was the bottleneck, we would expect the cost per word to halve when sending to two cores compared to one, be reduced by a third when sending to three compared to two, fall by a quarter between three and four, etc. However, Figure 5a shows that between one and two cores the cost is reduced by 50%. Likewise, between two and three it is reduced by 33%. Sending to any more cores simultaneously does almost not change the cost per word. The potential

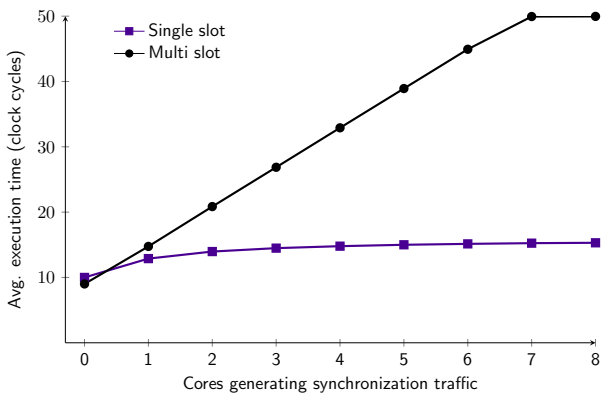


(a) Interprocessor communication execution times per word for the SSPM compared to the Argo NoC, where one core is simultaneously sending to between one and eight receiving cores. Synthetic scenario with a write, send, read, and acknowledge.

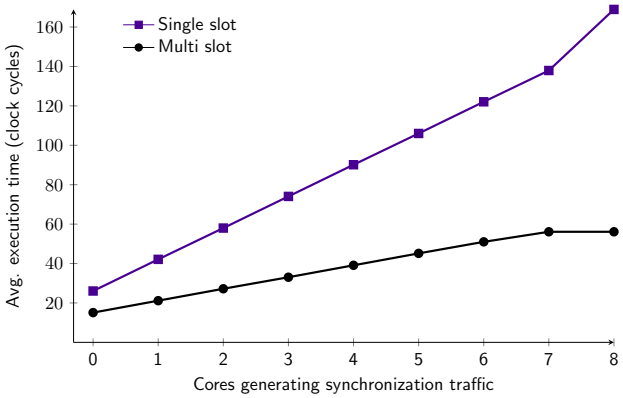


(b) Interprocessor communication execution times per word for the SSPM compared to the Argo NoC, where between one and four pairs of cores are communicating in parallel. Synthetic scenario with a write, send, read, and acknowledge.

Figure 5: Parallel channels.



(a) Average execution time to perform a write under synchronization traffic.



(b) Average execution time to perform synchronization under synchronization traffic.

Figure 6: Results with synchronization traffic for the SSPM.

bandwidth of the Argo NoC is therefore not completely utilized, as the sending core's copy-loop limits that performance. This could either be caused by the core itself being too slow, or by the bandwidth between the core and its local NoC scratchpad. Increasing the performance of the true culprit is the only way to unlock more of the NoC's bandwidth.

As the last bandwidth experiment, we explore true parallel independent channels. Figure 5b shows the result of the same end-to-end example with between one and four cores and sending to between one and four receivers. This is different from the previous figures in that each sender is only sending to one receiver. The figure shows that SSPM is more efficient than the NoC like we saw with only one channel. This is because the TDMA schedule effectively has those four parallel channels since each core can read or write once in a round.

It has to be noted that with a relative low core count a simple bus (or on-chip multiplexed arbitration) is more efficient than a NoC. With a higher numbers of cores we assume that a NoC solution provides better scalability. We plan as future work to port the T-CREST platform to a high-end FPGA platform where we can experiment with up to 64 cores.

5.5. Synchronization Support

In this subsection, we explore the difference between the two version of synchronization support with an extended access slot for atomic memory operations. Figure 6 shows the effects of extended time slot traffic on the performance in each of the SSPM implementations. The values are the averages of 1000 measurements.

Figure 6a shows the average execution time of a write to the SSPM while other cores are executing extended time

slot requests. We see that for the single slot implementation, the number of cores performing extended time slot traffic only has a slight influence on the write time. For more requests to the extended time slot, the TDM round is just slightly lengthened. Therefore, the influence of normal memory operations is minimal and bounded. For the multi slot access, the arbitration round is considerably increased and we observe a linear increase of the access time.

Figure 6b shows the average execution time of an atomic locking sequence, which uses extended time slots. We see that for both implementations the time to acquire a lock, which is assumed to be free, increases linearly with traffic, though the single slot implementation's execution time increases much faster than the multi slot's.

We see from these results that the choice of implementation should be on the basis of expected use case. The multi slot SSPM is superior in scenarios where many atomic operations are needed, while the single slot implementation should be used for scenarios where atomic operations are much rarer than normal memory operations.

From these results, we can also envision another design alternative: splitting the SSPM into two memories: one for normal data and one just for synchronization support. The latter one can then be used in multi slot mode for locks to protect data allocated in the normal shared scratchpad memory.

5.6. Reproducing the Results

As we are working in the context of an open-source project, it is relatively easy to provide a description of how to reproduce the presented results. We think that this possibility to (relatively) easy reproduce our presented results gives our contribution a stronger creditability.

The T-CREST project is open-source and the README[1] of the Patmos repository provides a brief introduction how to setup an Ubuntu installation for T-CREST and how to build T-CREST from the source. More detailed installation instructions are available in the Patmos handbook [19].

The T-CREST setup supports as a default target the DE2-115 FPGA board,[2] which we used for the implementation and evaluation. Although, the SSPM is independent of the FPGA type.

The implementation of the SSPM is available in the master branch of the T-CREST repositories. The hardware itself is in the Patmos repository. For the setup of the multicore platform Aegean has been changed accordingly. The SSPM is included in the default build for the multicore. For the evaluation we used the 9-core setup of T-CREST. The implementation of the SSPM itself is open source and includes a description on how to build the concrete multicore with the SSPM and how to run the benchmarks. Those instructions how to build and evaluate the SSPM can be found in the README at: `https://github.com/t-crest/patmos/blob/master/c/apps/sspm/README.md`.

[1] `https://github.com/t-crest/patmos`

[2] Available from: `http://de2-115.terasic.com/`

6. Conclusion

In this paper, we presented a solution for time-predictable communication between processor cores on a multicore processor. A shared scratchpad memory with synchronization support provides support for the data exchange via the memory and protection of that shared data with the synchronization support. The synchronization support is implemented by bus locking using extended time slots of the time-division multiplexed arbiter. This extended time slot allows for implementing atomic operations on a processing core without further modification needed. Using time-division multiplexing for the access to the shared scratchpad memory provides a time-predictable communication solution for hard real-time systems. The worst-case execution time for operations in the extended time slot can be bounded and therefore supports time-predictable locking.

Acknowledgment

The work presented in this paper was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project PREDICT,[3] contract no. 4184-00127A.

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[3] William J. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *DAC*, pages 684–689. ACM, 2001.

[4] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Conference on Design, Automation and Test in Europe*, DATE '14, pages 97:1–97:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[5] Henrik Enggaard Hansen, Emad Jacob Maroun, Andreas Toftegaard Kristensen, Jimmi Marquart, and Martin Schoeberl. A shared scratchpad memory with synchronization support. In *2017 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–6, Oct 2017.

[6] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. [Online, last accessed November 2013].

[7] Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, and Peter P. Puschner. The platin tool kit - the T-CREST approach for compiler and WCET integration. In *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

[3] `http://predict.compute.dtu.dk/`

[8] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 2. 2016.

[9] Mikyung Kang, Eunhui Park, Minkyoung Cho, Jinwoo Suh, Dong-In Kang, and Stephen P Crago. MPI performance analysis and optimization on Tile64/Maestro. In *Proceedings of Workshop on Multi-core Processors for Space—Opportunities and Challenges Held in conjunction with SMC-IT*, pages 19–23, 2009.

[10] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christian T. Müller, Kees Goossens, and Jens Sparsø. Argo: A real-time network-on-chip architecture with an efficient GALS implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24:479–492, 2016.

[11] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory. *ACM SIGPLAN Notices*, 28(7):54–63, 1993.

[12] Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob Van Der Wijngaart. *The Case for Message Passing on Many-Core Chips*, pages 115–123. Springer New York, New York, NY, 2011.

[13] Leslie Lamport. New solution of Dijkstra's concurrent programming problem. *Commun Acm*, 17(8):453–455, 1974.

[14] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–88. IEEE Computer Society, 2004.

[15] Matteo Monchiero, Gianluca Palermo, Cristina Silvano, and Oreste Villa. Exploration of distributed shared memory architectures for noc-based multiprocessors. *Journal of Systems Architecture*, 53(10):719–732, 2007.

[16] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. Compiling for time predictability. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 382–391. Springer Berlin / Heidelberg, 2012.

[17] Jack Sampson, Rubén González, Jean-Francois Collard, Norman P. Jouppi, and Mike Schlansker. Fast synchronization for chip multiprocessors. *ACM SIGARCH Computer Architecture News*, 33(4):64, November 2005.

[18] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[19] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.

[20] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Feb 2018.

[21] Martin Schoeberl and Jens Sparsø. Timing organization of a real-time multicore processor. In *2017 New Generation of CAS (NGCAS)*, pages 89–92, September 2017.

[22] Rasmus Bo Sørensen, Wolfgang Puffitsch, Martin Schoeberl, and Jens Sparsø. Message passing on a time-predictable multicore processor. In *Proceedings of the 17th IEEE Symposium on Real-time Distributed Computing (ISORC 2015)*, pages 51–59, Auckland, New Zealand, April 2015. IEEE.

[23] Christian Stoif, Martin Schoeberl, Benito Liccardi, and Jan Haase. Hardware synchronization for embedded multi-core processors. In *Proceedings of the 2011 IEEE International Symposium on Circuits and Systems (ISCAS 2011)*, Rie de Janeiro, Brazil, May 2011.

[24] Antonino Tumeo, Christian Pilato, Gianluca Palermo, Fabrizio Ferrandi, and Donatella Sciuto. HW/SW methodologies for synchronization in FPGA multiprocessors. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, page 265, New York, USA, 2009. ACM Press.