A Shared Scratchpad Memory with Synchronization Support

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

Published in:
Proceedings of the IEEE NorCAS 2017

Publication date:
2017

Document Version
Peer reviewed version

A Shared Scratchpad Memory with Synchronization Support

Henrik Enggaard Hansen, Emad Jacob Maroun, Andreas Toftegaard Kristensen,
Jimmi Marquart and Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark, Kgs. Lyngby
Email: [s134099, s123791]@student.dtu.dk, atkris@dtu.dk, s134111@student.dtu.dk, masca@dtu.dk

***Abstract*—Multicore processors usually communicate via shared memory, which is backed up by a shared level 2 cache and a cache coherence protocol. However, this solution is not a good fit for real-time systems, where we need to provide tight guarantees on execution and memory access times.**

In this paper, we propose a shared scratchpad memory as a time-predictable communication and synchronization structure, instead of the level 2 cache. The shared on-chip memory is accessed via a time division multiplexing arbiter, isolating the execution time of load and store instructions between processing cores. Furthermore, the arbiter supports an extended time slot where an atomic load and store instruction can be executed to implement synchronization primitives. In the evaluation we show that a shared scratchpad memory is an efficient communication structure for a small number of processors; in our setup, 9 cores. Furthermore, we evaluate the efficiency of the synchronization support for implementation of classic locks.

I. Introduction

Multi-processor systems-on-chip have become the standard for modern embedded systems. In these systems, interprocessor communication is essential to support parallel programming models. For this, we can use the message passing model, where each processor has its own private memory and communicates with other processors by sending and receiving messages. The shared memory model is an alternative to this. Here, the cores communicate by reading and writing to a shared memory, accessible to all processors. The region into which cores write and read to then determine which cores are communicating.

Furthermore, 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. This ensures that accesses to shared resources, such as a shared memory, are correctly sequenced and mutually exclusive during the execution of critical sections of an application. Furthermore, when such embedded systems have timing requirements, such as real-time systems, 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. Many soft-core processors, however, do not support atomic instructions, and thus other synchronization mechanisms are required to allow cooperation of the processors.

In this paper, we investigate the use of a shared scratchpad memory with hardware support for synchronization and interprocessor communication. The proposed solution is integrated into the hard real-time T-CREST multicore platform [13], where atomic primitives are not available in the cores that compose the system. Our objective is to provide efficient and time-predictable synchronization primitives and support interprocessor communication by utilizing the arbitration scheme for the shared scratchpad memory. All of this is done solely by peripheral hardware to the processing core and use of the bus protocols.

Since we integrate into a real-time system, we analyze the WCET of the implemented solution. Furthermore, we present the average-case execution times for interprocessor communication, using both the shared scratchpad memory and the Argo NoC of the T-CREST platform implemented on the Altera Cyclone EP4CE115 device.

The contributions of this paper are: (1) A shared scratchpad memory with synchronization support integrated into the T-CREST platform, and (2) an evaluation of the shared scratchpad memory and a comparison with the Argo NoC.

This paper is organized in six sections: Section II presents work related to hardware mechanisms for synchronization support and interprocessor communication. Section III provides background on the T-CREST platform. Section IV presents the design and implementation of the shared scratchpad memory with synchronization support. Section V presents the results of the implemented solution and evaluates it against the Argo NoC. Section VI concludes the paper.

II. Related Work

The use of hardware for lock-based synchronization mechanisms is explored in [17]. 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.

In IA-32 bus locking through the LOCK instruction guarantees the atomic execution of other instructions, e.g. compare-and-swap [5]. Bus locking reserves the memory bus for subsequent instructions allowing for the compare-and-swap instruction to complete in a globally consistent manner.

978-1-5386-2844-7/17/$31.00

Many soft-core processors such as the Xilinx MicroBlaze and the Patmos processor do not support atomic instructions for synchronization. The authors in [18] present hardware implementations of basic synchronization mechanisms to support locking and barriers for the soft-core Xilinx MicroBlaze processor.

In [12] a hardware mechanism to control processor synchronization is presented. This work offers fast atomic access to lock variables via a dedicated hardware unit. When a core fails to acquire a lock, its request is logged in the hardware unit. When the lock is released, an interrupt will be generated for the processor. For interrupts, the system supports both FIFO and priority-based schemes.

Regarding message passing in a time-predictable network-on-chip (NoC), the authors of [16] 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.

In [1], the authors suggest a hybrid message passing and shared memory architecture using the cache. They introduce a new cache line state, *possibly-stale*, using a conventional coherence protocol. Data is moved between nodes without the overhead of coherence, while at the same time keeping caches coherent to provide a traditional shared memory model.

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 [8]. 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 the comparison the performance is equivalent. They present an architecture supporting both mechanisms for interprocessor communication.

In [6] an implementation of MPI 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. The method presented in this paper avoids this by using the shared scratchpad memory for communication.

III. The T-CREST Platform

The T-CREST multicore platform is a multi-processor hard real-time system designed with the goal of having a low WCET and to ease its analysis [13]. All components have thus been designed with a focus on time-predictability, 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 [7] and a shared memory access NoC called Bluetree [2].

A processing node consists of a RISC processor called Patmos [15], special cache memories and a local scratchpad memory. Patmos is supported by a compiler developed with a focus on WCET [11], based on the LLVM framework [10]. The compiler can work with the aiT [3] tool from AbsInt and the open-source tool platin [4], which allows static computation of WCET bounds and supports the specific architecture of Patmos.

The Argo NoC [7] 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 (DMA) 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 static time-division multiplexing (TDM) scheduling for routing communication channels in routers and on links. The network interface between the NoC and the processor node integrates the DMA controllers with the TDM scheduling such that flow control and buffering are unnecessary.

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 [16]. Apart from the NoC, it is possible to implement software based methods for synchronization in T-CREST. The data cache can be invalidated [14] and Lamport's bakery algorithm [9] can be used to implement locks.

IV. Design and Implementation

The shared scratchpad memory presented in this paper is structured with an interface for each processor and an arbiter, as shown in Fig. 1. From the perspective of a single processor core the shared scratchpad memory behaves as a conventional scratchpad on a peripheral device bus with conventional read and write commands.

The *core interfaces* keeps read and write requests (referred to as *commands*) in a buffer and handles the protocol with a core associated with each core interface.. The *arbiter* multiplexes access to the memory using TDM with time slots, ensuring both evenly distributed access for all cores and also arbitration of concurrent access. Only within its time slot can a respective core fulfill its read or write commands.

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 has been received; by withholding this response the shared scratchpad memory can control the influx of commands from cores. When the time slot arrives, the arbiter allows for the fulfilment the command and issues a 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 shared scratchpad memory.

A. Atomicity Through Bus Locking

Support for atomic operations, such as acquiring a lock, is implemented by granting exclusive access to a single core for multiple cycles. This allows for the fulfillment of reads and writes on the shared scratchpad memory without interference

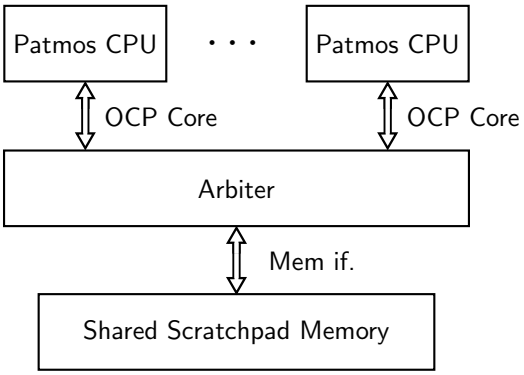


Fig. 1. The top level design of the shared scratchpad memory.

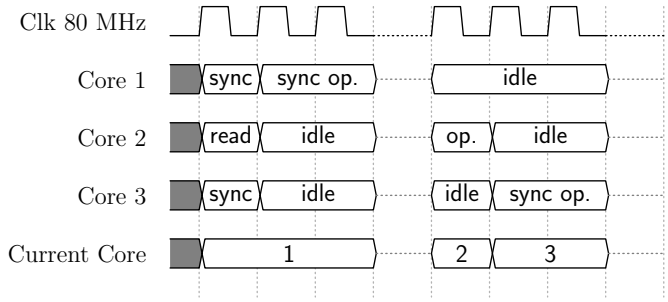


Fig. 2. An example showing the arbiter behavior when multiple cores request an action simultaneously. When cores are waiting, they go idle until their time slot arrives.

from other cores, thus mimicking the global behavior of atomic operations.

A core can request such an extended time slot from the shared scratchpad memory by issuing a read request for a special address. When the core's time slot arrives, the arbiter grants an extended time slot to allow for fulfilling multiple commands consecutively.

In Fig. 2, core 1 and core 3 request extended time slots (`sync`) and core 2 has an outstanding `read`. The current time slot ("Current Core") is for core 1 and the extended time slot is granted immediately. 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.

A test-and-set based locking mechanism fits this scheme, using 3 instructions:

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

The extended time slot is requested in the first load, and when the extended time slot arrives, the second load and the store can be served uninterrupted. If the load of `&lock` returned a 1 (locked), then the sequence of instructions changed nothing and the processor did not acquire the lock.

The shared scratchpad memory includes a buffer to meet timing specifications and commands are thus always delayed by one cycle. A minimum of 6 cycles is thus needed in the extended time slots to guarantee atomicity.

The extended time slots will influence the worst-case and average-case performance of loads and stores in the shared scratchpad memory. 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. Thus, the worst-case delay for a command, be it load, store, 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.

B. 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 commands should be shorter. These two assumptions allow for an alternate design for the arbiter.

When read or write commands are handled, the shared scratchpad memory 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 "round" has passed. By a "round" it is meant that every core must get a chance to fulfill a normal read and write, before a new extended time slot is granted.

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 lowered 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.

V. Evaluation and Results

This section presents the experimental evaluation of the shared scratchpad memory. All results of our architecture were produced using Altera Quartus (v16.1), targeting the Altera Cyclone FPGA (model EP4CE115) with Quartus' optimization

TABLE I
WORST CASE DELAYS (WCD) FOR ACCESS TO THE SHARED SCRATCHPAD MEMORY IN CLOCK CYCLES. *ETS* INDICATES THE DELAY FROM REQUESTING AN EXTENDED TIME SLOT TO HAVING IT ALLOCATED.

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

mode set to "Performance (Aggressive)". The clock frequency was set to 80 MHz as this is the limit for the Patmos core in this FPGA. A 9 core design was implemented where the shared scratchpad memory was added to the default T-CREST implementation containing the Argo NoC. Two such implementations were tested, one using the single slot shared scratchpad memory and the other for 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.

A. Worst-case delays

The worst-case delays can be seen in Table I, which also provides for an early comparison of multi slot and single extended time slot (single slot). It is immediately seen that the single slot mechanism significantly reduces the worst-case delay for reads and writes. However, this guarantee comes at the cost of the worst-case delay for an extended time slot. 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 [16] with a worst-case latency of 211 cycles for blocking transfer of two 4-byte words between two cores. The shared scratchpad memory occupies the core for 96 cycles with the multi-slot arbiter. Assuming the single slot arbiter, the worst-case delay is 26 cycles.

Comparing these results with the conclusion from [8] reveal that the shared scratchpad memory is positioned as a trade-off between shared-memory and message passing. Transfer latency is smaller than for message-passing, but there is no need for prefetching. However, the shared scratchpad memory includes a performance penalty which grows linearly with the number of cores and atomic operations are penalized further.

B. Resource Consumption

The resource consumption results are presented in Table II. The shared scratchpad memory solutions presented are very cheap, resource-wise, compared to the Argo NoC and the Patmos CPU. Compared to the arbiter and controller for main memory, our solution is also relatively cheap. Furthermore, 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. We do not show the size of the scratchpad memory itself, as it is configurable.

TABLE II
THE UTILIZATION OF THE WHOLE SYSTEM (ALL 9 CORES).

Entity	LUTs	Flip-Flops	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	-

C. Communication

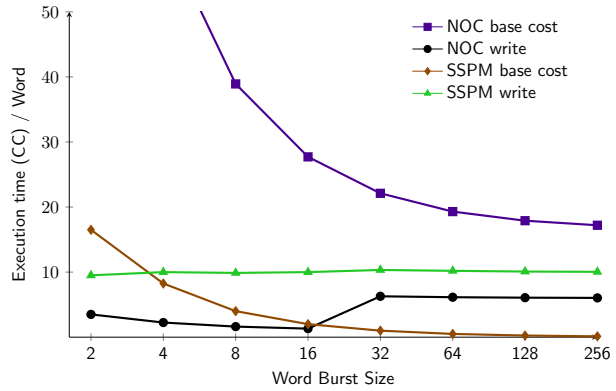
Figures of interest are write speed and roundtrip times. We do not benchmark read times to the scratchpad memories, since they are equal to the write times, which will be used instead. We define the roundtrip time as the time from the sender sends a message until it has observed an acknowledge from the receiver. The test was parametrized by the word size of the messages.

In Figure 3a, we show the execution times of writes to the Argo NoC local scratchpad and the shared scratchpad memory. The write speed of the NoC is greater than the shared scratchpad memory. This is expected, as the shared scratchpad memory's time multiplexed access will incur a performance hit for raw write speed. At a message size of 32 words, the NoC shows a performance decrease. This is the result of the compiler loop-unrolling the writes when the burst size is 16 or less, bypassing the loop overhead. Note that this is not done for the shared scratchpad memory writes for reasons unknown.

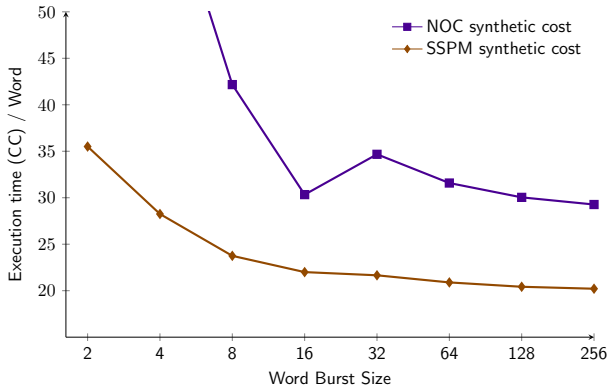
Figure 3a also shows the cost of a roundtrip using the Argo NoC and the shared scratchpad memory. The numbers demonstrate the advantage of a shared memory structure compared to a message passing NoC. The Argo NoC has a high fixed roundtrip overhead cost that scales with message size. With two word bursts, the average per word cost is ~ 106 cycles, falling to ~ 62 cycles with four word bursts. In contrast, the shared scratchpad memory's overhead is a fixed 34 cycles when no cores are using extended time slots, regardless of word size.

In Figure 3b we model a simple, synthetic application scenario: a core receives some input, works on it in some way, and then sends it on to another core. This scenario requires one read from a scratchpad, one write to a scratchpad, and then issuing a send to the next core. We ignore what work a core would do to the data since we are only interested in the cost of communication. In our test, the cores do no work. The figure shows the cost per word for the Argo NoC versus our shared scratchpad memory with the data extracted from Figure 3a. We use two writes, since writes and reads cost the same, and the base cost. The results show that our setup is more efficient at message passing than the Argo NoC.

The shared scratchpad memory solution does not scale as well as the Argo NoC. With an increase in core count the shared

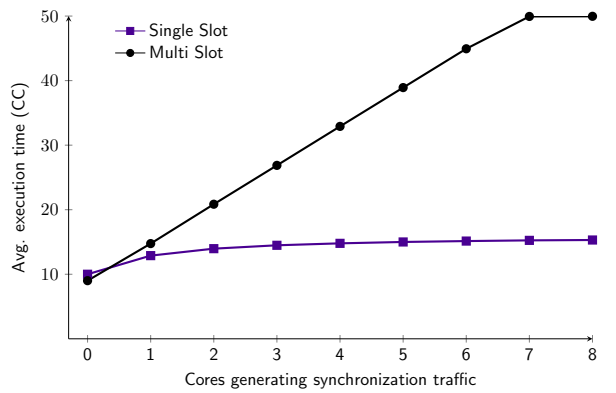


(a) Interprocessor communication execution times and scratchpad access time per word for the shared scratchpad memory compared to the Argo NoC. Base cost is the per word cost for sending the message until acknowledging that message.

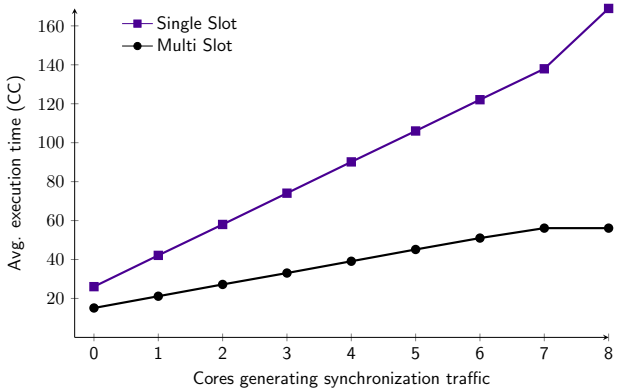


(b) Interprocessor communication execution times per word for the shared scratchpad memory compared to the Argo NoC. Synthetic scenario with a write, send, acknowledge, and read.

Fig. 3. Results for interprocessor communication.



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



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

Fig. 4. Results with synchronization traffic.

scratchpad memory 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. Additionally, the shared scratchpad memory is easier to program against, and arguably more flexible.

Conversely, the Argo NoC supports the same 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. The shared scratchpad memory solution needs to share bandwidth between all active channels.

D. Extended Time Slots

Figure 4 presents the effects of extended time slot traffic on performance in each of the shared scratchpad memory implementations. The values are the averages of 1000 measurements.

Figure 4a shows the average execution time of a write to the shared scratchpad memory 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. We would expect no change at all due to the TDM based arbitration, but, the initial lower execution time stems from the fact that if no extended time slots are requested, then none are granted. For the multi slot, write time increases linearly with extended time slot traffic.

Figure 4b 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 shared scratchpad memory is superior in scenarios where many atomic operations are needed, while the single slot implementation should be used for scenarios where atomic operations are a much rarer occurrence.

E. Reproducing the Results

We think reproducibility is of primary importance in science. As we are working in the context of an open-source project, it is relatively easy to provide pointers and a description of how to reproduce the presented results.

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, including setup on macOS, are available in the Patmos handbook [14]. To simplify the evaluation, we also provide a VM[2] where all needed packages and tools are already pre-installed However, that VM is currently used in teaching and you should reinstall and build T-CREST as described in the README.

The implementation of the shared scratchpad memory and the benchmarks are available in two forked repositories: (1) Patmos and (2) aegean. The implementation of the shared scratchpad memory itself is open source[3] and includes a description on how to build the concrete multicore with the shared scratchpad memory and how to run the benchmarks.

VI. Conclusion

In this paper, we presented a solution for time-predictable communication between processor cores on a multicore processor. We presented a shared scratchpad memory with support for bus locking using extended time slots. In turn, this allowed for implementing atomic operations on a processing core with no 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 this operation 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,[4] contract no. 4184-00127A.

References

[1] Matthew I. Frank and Mary K. Vernon. A Hybrid Shared Memory/Message Passing Parallel Machine. In *International Conference on Parallel Processing*, pages 232–236. IEEE, 1993.

[2] Jamie Garside and Neil C Audsley. Investigating shared memory tree prefetching within multimedia noc architectures. In *Memory Architecture and Organisation Workshop*, 2013.

[3] 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].

[4] 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.

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

[6] 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 SpaceOpportunities and Challenges Held in conjunction with SMC-IT*, pages 19–23, 2009.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] Bilge E. Saglam and Vincent J. Mooney III. System-on-a-chip processor synchronization support in hardware. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition*, pages 633–639. IEEE Comput. Soc, 2001.

[13] 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.

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

[15] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[16] 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.

[17] 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.

[18] 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.

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

[2] http://patmos.compute.dtu.dk/

[3] https://github.com/henrikh/patmos/tree/SSPM-device/hardware/src/sspm

[4] http://predict.compute.dtu.dk/