State-based Communication on Time-predictable Multicore Processors

Rasmus Bo Sørensen, Martin Schoeberl, Jens Sparsø
Department of Applied Mathematics and Computer Science
Technical University of Denmark
rboso@dtu.dk

ABSTRACT

Some real-time systems use a form of task-to-task communication called state-based or sample-based communication, where the read and write communication primitives do not impose any coordination among the communicating tasks. A reader may read the same value multiple times or may not read a given value at all. The concept is similar to a shared variable. This paper explores implementations of state-based communication in such network-on-chip based multicore platforms. We present two algorithms and outline their implementation. Assuming a specific time-predictable multicore processor, we evaluate how the read and write primitives of the two algorithms, contribute to the worst-case execution time of the communicating tasks.

Keywords

ACM proceedings; LaTeX; text tagging

1. INTRODUCTION

In real-time systems, a variation of channel-based communication that emphasizes modularity and encapsulation is often denoted by phrases and terms like *sending of state messages* [6] or *sample based* communication [1]. The semantics of these types of *state-based* communication resembles a shared variable that is accessed atomically by a single writer and one or more readers without any coordination. A value may be read multiple times or not at all before it is overwritten by the next value.

This paper designs and evaluates state-based communication algorithms for multicore systems, such that the communicating tasks executing on separate cores can be scheduled independently. The analysis of systems, where tasks executing on multiple cores can be scheduled independently, scales much better to many cores. We investigate two *state-based* communication algorithms on a time-predictable multicore platform with a network-on-chip (NOC) for inter-core communication. We analyze the worst-case latency of reading and writing a state message.

RTN'16 July 5th, 2016, Toulouse, France
© 2017 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

A time-triggered algorithm avoiding buffering and locking is possible, but requires a global schedule of the task execution and the inter-task communication. Our algorithms offer modularity through independent timing and schedulability analysis of the communicating tasks on individual cores. As shown in the results section the latency and the jitter introduced by the coordination among the writer and the readers is small.

We evaluate our work on the T-CREST platform, a multicore platform developed specifically to be time-predictable. However, the presented designs and analysis can easily be adapted to other multicore platforms that include a message passing NOC with guarantees on latency and bandwidth. An example of such a NOC is the Kalray MPPA processor series [4], which uses network calculus [11] to find guarantees

The contributions of this paper are: (i) three WCET-analyzable implementations of state-based communication that target and exploit NOC-based multicore platforms and aim at minimizing interference, (ii) an evaluation of the worst-case delay of communicating a state value between two tasks, for each implementation.

The paper is organized as follows: Section 2 provides related work on state-based communication and communicating tasks. Section 3 describes the system model and the hardware platform we use. Section 4 describes the three communication algorithms. Section 5 presents the analysis of the worst-case communication delay through a communication flow. Section 6 evaluates the presented designs. Section 7 concludes the paper.

2. RELATED WORK

The concept of state-based communication is equivalent to a shared variable that can be written by a single writer process and read by multiple reader processes. A more general version of this problem, allowing multiple writer processes, is called the *readers/writers problem*. This problem is defined by Courtois et al. [3] and later discussed in [9, 13], non of the presented solutions are suitable for true concurrent real-time systems.

In general-purpose systems, the predominant way of achieving mutual exclusion is to use a lock. As a solution to the readers/writers problem, read-write locks have been proposed by Mellor-Crummey and Scott [12]. Krieger et al. [7] propose a similar version of the read-write lock that requires fewer atomic operations. Real-time bound of the blocking time of these read-write lock are presented in [2].

This work evaluates how communication, previously mapped to off-chip shared memory, can be mapped to on-chip distributed memory communicating through a NOC. The evaluation is made with respect to hard real-time systems.

3. SYSTEM MODEL

This section describes the semantics of state-based communication, the application model, and the platform model and evaluation platform that we assume in our analysis of the problem and in the analysis of our proposed solutions.

3.1 State-based Communication

The concept of state-based communication is vaguely defined in the literature. The fundamental mechanism involves a writer and (possibly) multiple readers that operate without any coordination. The following sources from the real-time domain address the semantics of state-based communication.

The "The standard for space and time partitioned safety-critical avionic real-time operating systems" (ARINC 653) [1] describes two concepts for inter-partition communication; queuing ports and sampling ports. Queuing ports are similar to asynchronous message passing, and sampling ports are what we are concerned with in this paper.

In [6, sect.4.3.4] Kopetz discusses time-triggered messages – the alternative to event-based messages – and he writes: *"The semantics of state messages is similar to the semantics of a program variable that can be read many times without consuming it. Since there are no queues involved in state message transmissions, queue overflow is no issue. [...] State messages support the principle of independence [...] since sender and receiver can operate at different (independent) rates and there is no means for a receiver to influence the sender."*

In this paper we use the following semantics of state-based communication: State-based communication involves a single writer and one or more readers. Writing and reading of the state must be performed atomically. A read should always return the newest (completely) written state value, and at the global level, multiple concurrent readers should observe the same version of a state value at any given point in time. We refer to this as temporal consistency between readers. The writer and the readers will use an application programming interface and, in a real-time context, it is a requirement that the read and write primitives always complete execution in a bounded time; even when reading from an initially empty buffer.

3.2 Platform Model and Evaluation Platform

This paper considers a time-predictable multicore platform, where it is possible to derive tight WCET bounds for all hardware components. Fig. 1 shows a hardware platform where all processing cores are connected to a globally shared off-chip memory, and a network-on-chip that supports push communication between cores. Furthermore, we consider that each processing core has a local scratchpad memory (SPM), and that the network-on-chip can transfer data between the SPMs of the processing cores. The open-source multiprocessor T-CREST [15], which we use for evaluation, is such a platform.

To generalize our work and to avoid benchmarking features specific to the evaluation platform, we implement synchronization in software where needed. Implementing the communication primitives on a platform with hardware support for synchronization will allow optimization of the synchronization.

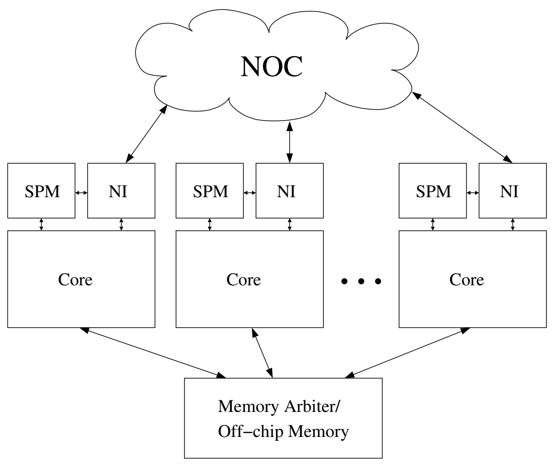


Figure 1: Block diagram of the assumed multicore processor. A processing core (Core), a scratch pad memory (SPM), and a network interface (NI) make up a tile connected via a network-on-chip (NOC). Each tile is also connected to the off-chip memory through a time-predictable memory arbiter.

3.3 Application Model

Our system contains a set $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ of n periodic tasks that communicate via a set $\mathcal{F} = \{f_1, f_2, ..., f_l\}$ of l state based communication flows.

Each communication flow $f_j \in \mathcal{F}$ is characterized by a tuple $(w_j, \mathcal{R}_j, m_j)$, where w_j is the task that writes to f_j, $\mathcal{R}_j$ is the set of tasks that read from f_j, and m_j is the size of a state message. It holds that $w_j \in \mathcal{T}$, $\mathcal{R}_j \subset \mathcal{T}$, and $w_j \notin \mathcal{R}_j$.

Each task $\tau_i \in \mathcal{T}$ is characterized by a tuple (T_i, C_i, p_i), where T_i is the period of the task, C_i is the WCET of the task, and p_i is the statically assigned processor core on which the task is executed on using partitioned scheduling. Each release of a task τ_i is called a job and may experience some release jitter J_i, introduced by the scheduling.

We consider multiple task running on each core, but only communicating tasks that execute on separate cores. Communicating tasks that execute on the same core can easily be added to any of the solutions we propose, because all our solutions work for true concurrent systems.

We assume that the reads and writes of communication flows, within a task τ_i, are executed unconditionally, such that the reads and writes are executed in the same sequence for every job. Therefore, each job is split into a sequence of phases. These phases can have one of three types: a reading phase, a writing phase or a computation phase. Thus, the total WCET C_i of a job can be decomposed into a sequence of phases, where each element of the sequence is the WCET of that execution phase.

We also assume that tasks holding a lock are not preempted by other tasks. This assumption reduces the inter-core interference when locks are used, and this simplifies the WCET analysis.

4. COMMUNICATION ALGORITHMS

This section describes common considerations of the two algorithms, followed by a description of each algorithm. The two algorithms can be implemented in off-chip shared memory or in on-chip distributed memory. We discuss the implications of the two implementations in the subsections of each algorithm.

4.1 Common Considerations

To ensure atomicity of the state-based communication, it is necessary that reads and writes are performed mutually exclusively, while concurrent reads are allowed. To guarantee mutual exclusion between the writer and the readers, the first two algorithms use a lock and the third algorithm uses a queue.

For the two algorithms that use a lock, the length of the critical section has a great impact on the synchronization delay experienced by each task. In this case, a lock needs to preserve the temporal ordering of the lock requests to ensure a solution to the readers/writers problem that is starvation free.

The temporal consistency property is fully satisfied for the two algorithms using a lock, because the lock enforces the strict ordering of accesses. Thus, all reads executed after a write will read the new value. We discuss the temporal consistency property of the algorithm that uses a queue in Subsection 4.3.

4.2 Algorithm 1: Single Shared Buffer and a Lock

A common practice of implementing state-based communication uses a single buffer. The shared buffer can be implemented in both the off-chip shared memory or in the on-chip distributed SPMs. In the off-chip shared memory, we allocate a buffer and protect it by a lock. In the on-chip processor-local SPMs, we implement the shared buffer by allocating it in the processor-local SPM of the reader. The read and write operations behave the same in both implementations. The write operation acquires the lock and transfers the new state value to the allocated buffer and then it releases the lock. The read operation acquires the lock and reads the newest state value from the allocated buffer, before it releases the lock.

This is probably the simplest implementation. However, we expect that the implementations have a long critical section because of the data copying in the critical section.

4.3 Algorithm 2: Message Passing Queue

A message passing queue can be used as a solution to the readers/writers problem, if we can find the upper bound on the number of elements in the queue that are needed to avoid overflow. With a queue, the writer writes to the next free buffer and the reader can read the newest value by dequeuing all available elements, only keeping the most recent one.

To find the upper bound on the number of elements that are needed to avoid overflow, we need to know the maximum write rate and the minimum read rate.

For periodic tasks, the rate is the number of writes or reads of the state value during the task period. Sporadic tasks are typically tasks triggered by external events. Therefore, sporadic tasks might produce new state values, but they are not typical state consumers. The production rate of a sporadic task is the number of produced state values over the minimum inter-arrival time. The number of elements needed in the queue is the ratio of the production rate over the consumption rate plus one extra buffer to account for phase alignment.

A drawback of the message passing queue is that the memory footprint increases with the period ratio of the writer and reader. When the reader has a shorter period than the writer, we only need two buffers at the reader side. Otherwise, more buffers are needed at the reader side. For large message sizes and a slow reader, this buffering scheme might not be practical.

In the shared-memory case, we use non-blocking single reader/single writer queues [10]. One queue to send a copy of the state and a return queue to return free buffers from the reader to the write for reuse. In the distributed memory case, we use the non-blocking single-reader/ single-writer queues [16]. The queue supports acknowledgments for reuse of buffers.

5. WORST-CASE COMMUNICATION DELAY

To ensure that communicating tasks executing on separate cores can be scheduled independently, we define the worst-case delay D_j of state-based communication flow f_j as the maximum separation time between the start of the write primitive and the end of the first instance of the read primitive that reads the new value. This definition makes the worst-case communication delay (WCCD) independent of the actual implementation.

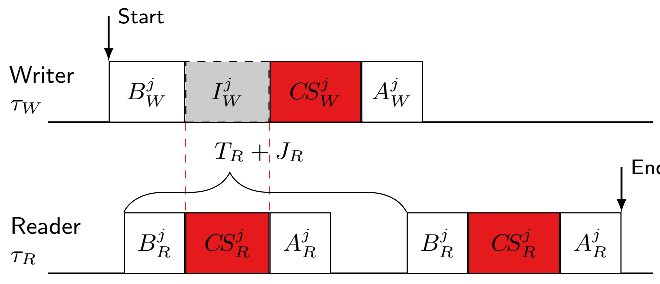
The worst-case alignment of the reader and writer tasks happen when the reader causes the maximum interference on the writer, which postpones the read of the new value to the next job of the reader task. An intuition on why this is the WCCD: If we shift the reader forward in time, the reader will be blocked by the write primitive and the reader will read the new value before it finishes. If we shift the reader backwards in time, the reader will block the writer for a shorter amount of time and the reader will read the new value sooner.

We present the WCCD formulas for the presented designs in the following two subsections. If the WCCD does not depend on the period of the writer, the writing task may be sporadic with a known minimum inter-arrival time.

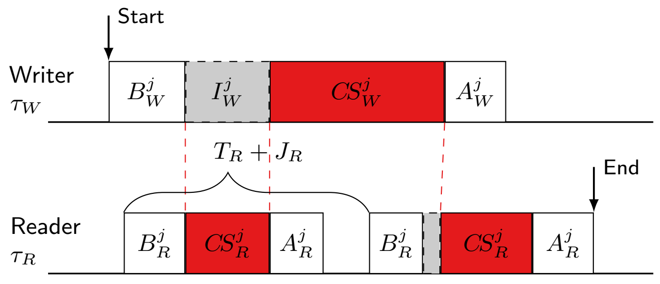
5.1 Algorithm 1: Shared Buffers and a Lock

For the two presented designs that use a lock to protect one or multiple shared buffers, we model the read and the write primitives, communicating through flow f_j, as five variables: (1) B_S^j is the WCET of the preamble before the critical section, (2) I_S^j is the worst-case synchronization interference, (3) CS_S^j is the WCET of the critical section, (4) A_S^j is the WCET of the postamble after the critical section, and (5) m_j is the message size of the state-based value. The subscript S of the variables denotes the index R of the reader task τ_R or the index W of the writer task τ_W. The superscript j of the variables denotes the flow index. The read and write primitives inherit their period T_i from the calling task. We add the release jitter of the read and write primitives to J_i of τ_i to shorten the expressions.

These variables decide which of the two scenarios in Fig. 2 occur. We find the WCCD in the two scenarios by adding the variables on the path from start to end. The path in scenario one is (B_W^j, $-B_R^j$, T_R, J_R, B_R^j, CS_R^j, A_R^j). Observe that if B_R^j of the first reader job is decreased, the WCCD is increased, because it moves the starting time of period and the second



(a) Scenario one



(b) Scenario two

Figure 2: The two scenarios that lead to the worst-case communication delay. In both cases the condition that leads to the worst-case is when reads and writes are aligned exactly so that the critical section of a read blocks the critical section of a write.

job forward in time. Therefore, the WCCD happens when the B_R^j of the first job is the best-case execution time (BCET) of the reader preamble $\hat{B}_R^j$. We show the WCCD for scenario one:

$$D_j = B_W^j - \hat{B}_R^j + T_R + J_R + B_R^j + CS_R^j + A_R^j \qquad (1)$$

We assume that $\hat{B}_R^j \ll T_R$, therefore we set $\hat{B}_R^j = 0$. This is a safe underestimation of the BCET that leads to a safe overestimation WCCD. We also find the path in scenario two and by reordering to resemble (1), we get:

$$D_j = B_W^j + I_W^j + CS_W^j + CS_R^j + A_R^j \qquad (2)$$

To unify the (1) and (2) we take the maximum of the two formulas. We show the formula for the WCCD:

$$D_j = B_W^j + \max(T_R + J_R + B_R^j, I_W^j + CS_W^j) + CS_R^j + A_R^j \quad (3)$$

The write primitive includes the complete transfer of the state value, the new value is available in the SPM of the reader task after the critical section of the writer. Therefore, the message size m_j changes the WCET of either the critical section CS_S^j or the preamble B_S^j, depending on which of the implementations that use a lock we choose. If we choose the single buffer implementation, increasing the message size m_j will increase the critical section CS_S^j. If we choose the three buffer implementation, increasing the message size m_j increases the preamble B_S^j.

In the single reader/single writer case, the worst-case synchronization interference I_W^j on τ_W from τ_R is equal to the length of the critical section of the reader CS_R^j. The worst-case synchronization interference I_R^j on τ_R from τ_W is equal to the length of the critical section of the writer CS_W^j

5.2 Algorithms 2: Message Passing Queue

The presented design that implements state-based communication with a message passing queue does not use a lock and therefore there is no synchronization interference. We model the read and write primitives for flow f_j as the WCETs Q_R^j and Q_W^j of the read and write primitive.

Compared to the two scenarios for the shared buffer and a lock, the message passing implementation only has one scenario. We show this scenario in Fig. 3. The WCCD for this scenario is:

$$D_j = Q_W^j + T_R + J_R + Q_R^j \qquad (4)$$

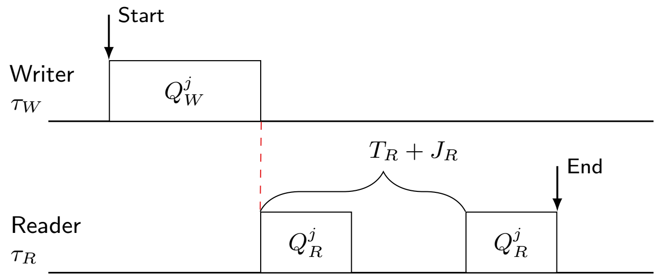


Figure 3: The scenario that leads to the WCCD for read and write functions that do not use a lock.

The value of Q_R^j depends on the number of elements that are needed in the queue to avoid overflow.

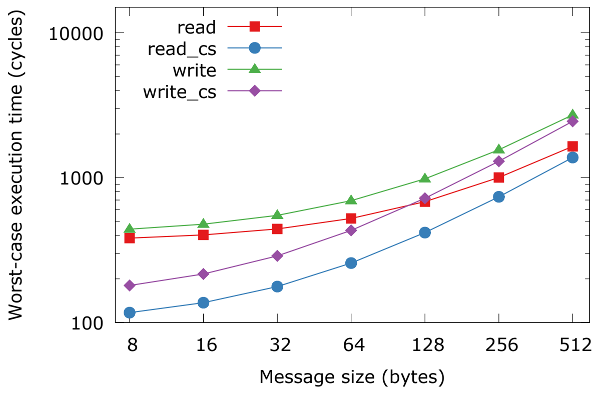
6. WORST-CASE EVALUATION

This section describes the evaluation setup and the evaluation of the three algorithms. As we consider real-time systems, we use static WCET analysis for the performance comparison instead of average-case measurements.

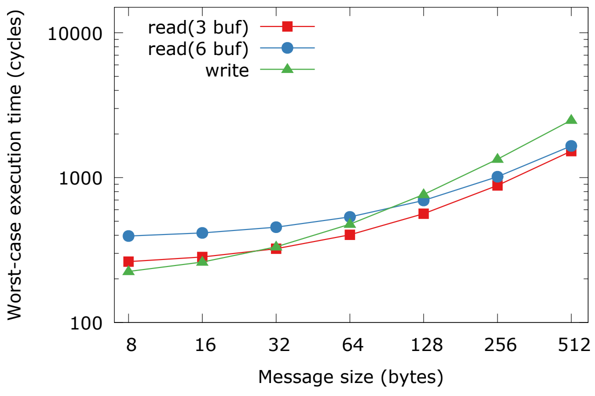
We evaluate all three algorithms on the distributed on-chip memory with the NOC. For the shared memory we evaluate only the the single buffer solution. As we see that this is an order of magnitude slower than distributed memory, we omit figures for shared memory for the other two solutions.

6.1 Evaluation setup

For this evaluation, we assume a 9 core platform, where the code for the primitives is stored in the instruction scratchpad memory. In the 9 core platform, the bandwidth towards main memory is divided equally between the 9 cores and the guaranteed-service of the NOC is setup such that all cores can send to all other cores with equal bandwidth. We refer to this NOC schedule as an all-to-all schedule. We find the WCET of the communication primitives with the aiT tool from AbsInt [5], which supports the Patmos processor. In the source code of the communication primitives, there are a number of `while` loops that wait until certain events that are time-bounded happen, such as the completion of a DMA transfer. The worst-case waiting time of a DMA transfer can be calculated based on the size of the transfer and the bandwidth of the communication flow towards the receiver. The worst-case wait time divided by the WCET of



(a) Algorithm 1



(b) Algorithm 2

Figure 4:

one iteration of the `while` loop is equal to the maximum loop bound of that `while` loop. For each configuration shown in the following plots, we find the WCET of one iteration of all the `while` loops. Based on the maximum waiting time and the WCET of each loop iteration, we can calculate the loop bounds of each `while` loop and pass them to the tool.

The WCET numbers that we show in the following subsections include the code for the locking functions `acquire_lock()` and `release_lock()`. To avoid benchmarking the locking, all implementations use the same lock to protect the shared data, regardless of whether they place data in shared off-chip memory or distributed on-chip memory. The lock that we use for the results is Lamport's Bakery [8] algorithm using the on-chip distributed memory. The Bakery algorithm is well-suited for implementation in distributed memory, because the variables can be laid out such that it uses local-only spinning and remote writes. If another locking method is used, the data points for the implementations using a lock will change by the same value in all the following plots. Therefore, choosing a different locking implementation will not change the results considerably.

The WCET of the lock is 240 cycles for the `acquire_lock()` and 82 cycles for the `release_lock()`. These numbers do not account for the interference from other cores that try to take the lock. A task is in a critical section from right after the `acquire_lock()` function returns until the `release_lock()` function returns, therefore we include the `release_lock()` in the critical section of the communication primitives. The application designer needs to add the interference of the other threads holding the lock to the length of the critical sections during the schedulability analysis.

6.2 Algorithm 1: Single Shared Buffer and a Lock

Fig. 4a shows the WCET of the read and write functions implemented using shared off-chip memory and Fig. 4b shows the WCET of the read and write functions implemented using distributed on-chip memory.

In Fig. 4a, we can see that, when the message size increases, the WCET of the shared-memory version becomes very high compared to the distributed-memory version. For the shared-memory version, we can see that the critical sections of the read and the write primitives are long and quickly become the dominating factor in the WCET.

For the distributed-memory version in Fig. 4b, we can see that the critical section of the write primitive is longer than the critical section of the read primitive. This is due to the fact that the network bandwidth of the all-to-all schedule is lower than the bandwidth between the local SPM and the processor.

For the read and the write primitives it is the locking overhead that causes the difference between the WCET of the critical sections and the overall WCET of the communication primitives. The difference between the lines for the critical sections and the line for the total WCET is constant across the message sizes.

6.3 Algorithm 2: Message Passing Queue

Fig. **??** shows the WCET of the write primitive and the WCET of the read primitive with 3 and 6 elements in the queue. The queuing implementation does not have any locking and therefore no critical sections. With 3 buffers, the implementation supports that the writer writes twice as fast as the reader reads. With 6 buffers the ratio is 5-times faster writes.

The number of buffers in the queue changes the WCET of the read primitive. In the worst-case, the reader needs to dequeue all the elements of the buffer and then return the last successfully dequeued message.

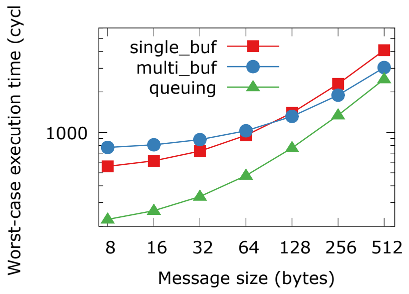
The WCET of the read primitive, increases with the number of buffers, but as the message sizes grow, the reading of the message becomes the dominating factor in the WCET.

6.4 Comparison

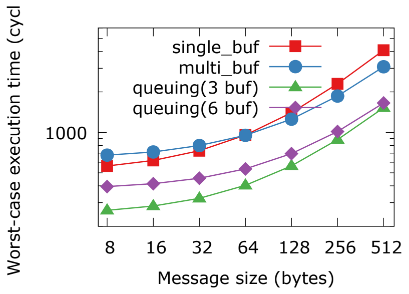
Fig. 5 shows the WCET of the write and read primitives in a system with one writer and one reader. The numbers for the write primitives include the worst-case synchronization interference from the read primitive and vice versa.

The queuing implementation does not suffer any synchronization interference, but the number of buffer elements in the queue depends on the ratio between the period of the reader and writer. These buffers are placed in the reader, and for each read the reader needs to dequeue all elements in the buffer. Therefore, we show plots for three and six buffers for the reader primitive.

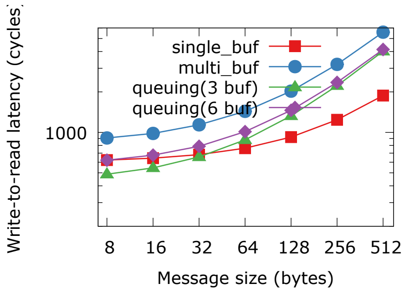
For any message size, the implementation of the queuing



(a) Write primitives



(b) Read primitives



(c) WCCD

Figure 5: The WCET of the read and write functions for all the presented implementations, as a function of the size of a sample, excluding the implementation using off-chip shared memory. WCCD excluding the period and jitter of the reader.

algorithm has a lower WCET than the implementations that use locks, even with six buffers in the reader queue.

Fig. 5c shows the WCCDs D_j minus the period T_R and the jitter J_R of our two solutions as a function of the message size, as shown in (3) and (4). We show the numbers without T_R and J_R because these variables are the same for the three solutions and they are determined by an application. For the WCCD of the two solutions using a lock, we assume that $T_R + J_R + B_R^j > I_W^j + CS_W^j$, which is scenario one from Fig. 2a.

7. CONCLUSION

This paper addressed the implementation of time-predictable state-based communication in multicore platforms for hard real-time systems. The concept of state-based communication is similar to a shared variable that can be written and read atomically.

Aiming for a solution that (a) scales better with a growing number of processors, and (b) has low latency and low jitter, this paper proposed and evaluated two algorithms that exploit the scalable message passing NOC and the processor-local memories found in many recent multicore platforms. The evaluation is based on actual hardware and the WCET and the worst-case communication delay in clock cycles, are obtained using the aiT tool from AbsInt.

The single-buffer algorithm has the lowest worst-case communication delay and it has a minimal memory footprint. For the single-buffer algorithm, the distributed memory and NOC implementation is around one order of magnitude faster than the shared memory implementation.

The implementation of the queuing algorithm has the lowest WCET across all message sizes, and if the ratio of the writer and reader periods is close to one, the memory footprint can be acceptable. The queuing algorithm has very low jitter because it suffers no synchronization interference.

Acknowledgment

The work presented in this paper was funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP [14], contract no. 12-127600. The work is open source and the full T-CREST tool chain can be downloaded from GitHub (https://github.com/t-crest/) and built under Ubuntu.

8. REFERENCES

[1] ARINC 653. Avionics application software standard snterface – Part 1: Required services, 2010.

[2] B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87, 2010.

[3] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with ”readers” and ”writers”. *Commun. ACM*, 14(10):667–668, Oct. 1971.

[4] B. Dupont de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 97:1–97:6, 2014.

[5] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

[6] H. Kopetz. *Real-Time Systems*. Kluwer Academic, Boston, MA, USA, 1997.

[7] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable reader-writer lock. In *Proc. Int. Conference on Parallel Processing (ICPP)*, volume 2, pages 201–204, 1993.

[8] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, Aug. 1974.

[9] L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, Nov. 1977.

[10] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[11] J.-Y. Le Boudec. Application of network calculus to guaranteed service networks. *Information Theory, IEEE Transactions on*, 44(3):1087–1096, May 1998.

[12] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 106–113. ACM, 1991.

[13] G. L. Peterson. Concurrent reading while writing. *ACM*

Trans. Program. Lang. Syst., 5(1):46–55, Jan. 1983.

[14] RTEMP project page, 2013. Available online at http://rtemp.compute.dtu.dk/.

[15] M. Schoeberl, S. Abbaspourseyedi, A. Jordan, E. Kasapaki, W. Puffitsch, J. Sparsø, B. Akesson, N. Audsley, J. Garside, R. Capasso, A. Tocchi, K. Goossens, S. Goossens, Y. Li, S. Hansen, R. Heckmann, S. Hepp, B. Huber, J. Knoop, D. Prokesch, P. Puschner, A. Rocha, and C. Silva. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[16] R. Sørensen, W. Puffitsch, M. Schoeberl, and J. Sparsø. Message passing on a time-predictable multicore processor. In *Proc. IEEE Int. Symposium on Real-Time Distributed Computing (ISORC)*, pages 51–59, April 2015.