Support for the Logical Execution Time Model on a Time-predictable Multicore Processor

Florian Kluge
Department of Computer Science
University of Augsburg
kluge@informatik.uni-augsburg.de

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
masca@dtu.dk

Theo Ungerer
Department of Computer Science
University of Augsburg
ungerer@informatik.uni-augsburg.de

ABSTRACT

The *logical execution time* (LET) model increases the compositionality of real-time task sets. Removal or addition of tasks does not influence the communication behavior of other tasks. In this work, we extend a multicore operating system running on a time-predictable multicore processor to support the LET model. For communication between tasks we use message passing on a time-predictable network-on-chip to avoid the bottleneck of shared memory. We report our experiences and present results on the costs in terms of memory and execution time.

1. INTRODUCTION

Modern multicore processors exhibit similarities to distributed systems. The cores on such processors are connected through *networks-on-chip* (NoCs), over which they can exchange messages. Increasing numbers of cores in modern processors enable the integration of new software features in *safety-critical system*s (SCSs). Initially, such new architectures posed some problems to developers of SCSs [19]. In the meantime, efforts have been made to overcome these problems by appropriate hardware and software design (see e.g. [24, 33, 34]). A time-predictable execution of tasks requires that both the cores and the NoC have a time-predictable behaviour. Furthermore, the *operating system* (OS) plays a key role in making the performance of multicore processors available to safety-critical applications in a predictable manner.

Another problem lies in the aspect of compositionality of such systems, a problem that is not restricted to multicores, but already exists for single-core processors. If, at some time during the development process, a new task is added to the system, the actual task schedules and important properties like task response times may change. If the design of application software depends on the respone times, the changes of the schedule may require that the application design is revised, as can be the case for control algorithms implemented in software. The concept of LET, as introduced in Giotto [13], provides a solution to this problem by fixing input and output times of tasks. Thus, compositionality of a task system is ensured, as long as schedulability can be guaranteed.

In this paper, we report our experiences on the integration of the LET execution model in a time-predictable multicore platform. As hardware we use the Patmos multicore processor [29] from the T-CREST project [27]. We port MOSSCA [21], a research prototype of a *manycore operating system for safety-critical applications*, to this platform and extend it by communication primitives for the NoC that are mapped into the LET model. The results reported in this paper give a first impression of the costs in terms of memory and execution time that can be expected from such extensions.

The remainder of this paper is structured as follows: Section 2 presents related work. Section 3 recalls the basic properties of the Patmos multicore processor, the NoC. The port of MOSSCA to the Patmos multicore processor is described in section 4. Section 5 introduces the extensions of MOSSCA to support LET. Quantitative results are reported in section 6. We conclude the paper in section 7.

2. RELATED WORK

Epiphany is a high-performance energy-efficient multicore processor [25] featuring a distributed memory architecture. Each core contains 32 KB of local memory that is mapped into a global address space. The processors contain no caches. Accesses to memory of a remote core is performed over a NoC. The NoC is organized as a mesh and favors writes over reads.

In contrast to Epiphany, our Patmos multicore processor contains a NoC with explicit support for message passing. Processor-local memories are only accessible from the local processor. Furthermore, Patmos supports, additionally to local memory, caches for instruction and data.

For time-predictable on-chip communication, a NoC with time-division multiplexing (TDM) arbitration allows bounding the communication delay. Æthereal [10] is such a NoC that uses TDM where slots are reserved to allow a block of data to pass through the NoC router without waiting or blocking traffic. Slot tables with routing information are contained in the routers and no arbitration or link-to-link flow control is required. Instead, a credit-based flow control is applied for end-to-end control, saving buffer space between links. aelite, a light version of Æthereal, only offers guaranteed services resulting in a simpler router design with source routing [11].

In contrast to the Æthereal family of NoCs our NoC implements TDM arbitration from end-to-end. I.e., access to the *scratchpad memory* (SPM) is scheduled with the NoC TDM schedule. Therefore we do not need any credit-based flow control.

RTN 2016, Toulouse, France

Theoretical approaches for calculating maximum response times of a NoC are network calculus [3, 4, 22] and response-time analysis [30]. Network calculus is used to compute bounds on buffer sizes and bounds on latencies. With traffic shaping, i.e., rate control at the message source, delays and buffer sizes can be bounded. The Kalray multicore processor [9] is especially designed to support time-predictable message passing with rate control in the sender and no further flow control [8]. The NoC response time analysis can be combined with task schedulability analysis to produce an end-to-end schedulability test for multicores [16].

When comparing TDM with network calculus [26], TDM results in shorter worst-case latencies while network calculus leads to higher bandwidth. However, TDM leads to simpler routers and network interfaces than support for a dynamically scheduled NoC. Therefore, a TDM based NoC is used in the Patmos multicore processor.

The time-composable operating system TiCOS [2] has been ported to Patmos [36]. TiCOS is a light-weight RTOS, which is a fork of the open-source POK project [7]. TiCOS conforms with ARINC 653 including the ARINC 653 events and ports for synchronization and inter-partition communication. In contrast to MOSSCA, TiCOS supports only a single processor core and therefore does not use the NoC for task communication.

3. BASELINE

3.1 Patmos

For safety-critical systems we need a time-predictable platform that allows for worst-case execution time (WCET) analysis. The Patmos processor [29] and the multicore version of Patmos developed within the T-CREST project [27] is designed to enable and simplify WCET analysis.

Patmos is a dual-issue RISC processor designed with the main objective to allow WCET analysis and is optimized for a low WCET bound. A main focus on Patmos is the memory hierarchy and the organization of local memories. For code Patmos contains a so-called method cache [6] and an *instruction scratch-pad memory* (ISPM). The method cache caches whole methods/functions to simplify WCET analysis. When caching whole functions all instructions except call and return are guaranteed hits. Cache misses can only happen on a call or return instruction. The method cache is integrated in the aiT [12] and platin [15] WCET analysis tools. The ISPM can be used for functions that shall be always loaded in a local memory, e.g., operating system services.

For data, Patmos contains a normal data cache, a stack cache [1], and a *data scratch-pad memory* (DSPM). The data cache is organized as write-through cache, as this is the form of data cache that is best supported by WCET analysis tools. Standard data caches hold data from different memory areas that are differently hard to analyze with respect to WCET. E.g., heap allocated data is practically impossible to analyze as addresses are not known statically for the analysis. However, stack allocated data is relatively easy to analyze. Therefore, stack data is cached in Patmos in the stack cache. Data that shall always be loaded in local memory, e.g., operating system data or data that shall be sent via the NoC, can be allocated in the data SPM.

3.2 Multicore Communication

The multicore version of Patmos is connected to two networks-on-chip (NOC): (1) a memory tree to access the shared external memory [28] and (2) the Argo message passing NoC [17]. The memory tree provides TDM arbitration to the main memory and is therefore time-predictable. However, communication between cores through main memory is very expensive as the memory is the bottleneck. The message passing Argo NoC is especially designed to allow time-predictable communication between processor cores.

Argo uses TDM for accessing the NoC and performs data movement from the sender SPM to the receiver SPM. A message is sent via the Argo NoC by a DMA inserting packets into the NoC. The novelty in Argo is that the DMA is shared by all message channels via TDM that is synchronous to the NoC TDM [32]. This mechanism allows for end-to-end TDM based transfer of the messages without any buffering (except pipeline registers) and flow control. Therefore, the worst-case transfer time can easily be computed [31].

3.3 MOSSCA

The design of MOSSCA is inspired by the factored operating system (fos) [35]. Its central concept is to distribute functionalities over a multicore processor, e.g., by assigning each application or thread a separate core. All communication is strictly based on explicit messages that are sent over the NoC. The implementation of MOSSCA assumes a multicore processor where each core possesses local memories where OS and application code and data can be stored. For communication, the NoC connecting the cores must be able to provide hard real-time guarantees.

For the application developer, MOSSCA provides three abstractions, namely *nodes*, *communication channels*, and *servers*. *Nodes* represent the primary execution resources for applications. Each MOSSCA node maps directly to a physical node of the multicore processor. A MOSSCA node acts as container for the binary images of applications and executes the program defined in this image. *Communication channels* represent the basic means for interaction between threads. A communication channel in MOSSCA provides unidirectional communication between a sender *(channel source)* and a receiver node *(channel sink)*. Channels can be used to implement more sophisticated communication patterns on top. A *channel policy* defines limits on the usage of channels to prevent overload of the NoC and the channel sink. MOSSCA *servers* provide services that are used by multiple applications or threads but need to be executed in a centralized manner. Examples are the multiplexing of I/O devices, or OS services that may afflict multiple nodes and therefor must be centralized. Also, applications may define servers that provide library services for computations used in several threads. MOSSCA servers implement non-interruptible transactions. Once a server has started processing an application request, it will finish this request without interruption by other application requests.

Figure 1 gives an overview of the MOSSCA system architecture. The hardware base is formed by *nodes*. These are connected by a *real-time interconnect* that provides predictable traversal times for messages. Additionally, some nodes have special facilities, e.g. for off-chip I/O. OS functionalities are split into several parts to run in a distributed manner and to achieve high parallelism. An identical kernel on each node is responsible for configuration and management of the node's hard- and software. OS functionalities that require a centralized execution or need to coordinate between several nodes are provided by OS servers. Nodes possessing an off-chip connection that is used by at least one of the applications act as I/O servers. For inter-partition communication, MOSSCA allocates dedicated communication servers. Additionally, MOSSCA supports the implementation of application-specific servers that provide e.g., library services used by several threads. Stubs on application nodes provide convenient methods to issue requests to servers.

In the actual implementation of MOSSCA, two kernel images are generated: The *boot kernel* is loaded first to a dedicated node

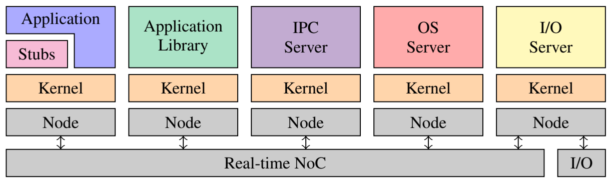

Figure 1: MOSSCA System Architecture

and coordinates the startup of the multicore processor. The *standard kernel* runs on any other node, it is loaded via the boot kernel. Loading of the kernel is managed by a small BIOS that is executed when the processor starts operation. For each application, a separate image is generated. The advantage of this approach is that the (standard) kernel image must be loaded from ROM only once and then can be distributed to all cores, thus speeding up the boot process [20]. The MOSSCA reference implementation is developed to run on a multicore simulator [23] that implements several instruction set architectures.

4. MOSSCA ON PATMOS

MOSSCA was ported to the Patmos multicore implementation with four cores for the Altera DE2-115 FPGA board. An implementation with nine cores is also available for the same board. In both implementations of the multicore processor, core 0 is connected to serial port of the board. The serial port is used both for loading of kernel and application images, and for text output. Requests for code images or output from other cores must be mediated by core 0.

The main challenge in porting MOSSCA to Patmos was the assignment and use of address spaces for the different parts of the kernel and application images. For our first implementation, we decided to leave the local SPMs free to be used by applications, as we expect the amount of application code executed to be much larger than the amount of OS code. Therefore, the MOSSCA kernel is placed in global memory. Kernel code exists only once and is used by all cores. For each core a separate kernel data section is allocated in global memory. Applications may use both global memory and the local SPMs for code and data.

To improve kernel performance, we performed a second implementation where kernel code is placed in the local instruction SPM (ISPM). In this implementation, application code must be executed from global memory, as most of the ISPM is taken by the kernel, and the ISPM could not be enlarged further due to synthesis restrictions. Kernel data is still kept in partitions of the global memory, while the DSPM is fully available to applications.

Communication between cores is performed using the message passing libraries from the T-CREST project [31]. MOSSCA channels are mapped directly to message passing channels defined in the libraries. These use a DMA engine to transfer data between the communication SPMs of two cores. Therefore, the data must be copied from the data SPM to the communication SPM.

In both implementations, the application running on core 0 on top of the boot kernel has two responsibilities: First, it loads and distributes the kernel and application images of all other nodes during bootstrapping of the processor. Second, during regular operation, it acts as a I/O server for access to the serial port of the board. Output routines for debugging (`printf` etc.) on other cores write their output data to buffers that are located in the global memory. The I/O server takes completed output messages from the buffers and sends them over the serial line to the connected computer.

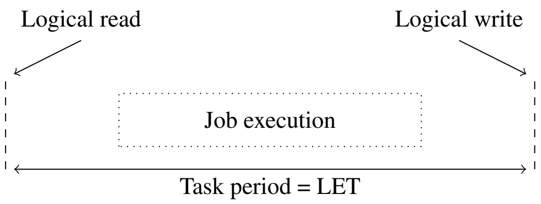

Figure 2: Logical execution time

5. LET SUPPORT IN MOSSCA

LET, as introduced in Giotto [13], abstracts from the physical execution of periodic tasks. The concept is illustrated in Figure 2. A task's LET is equal to its period. Input data is logically read at the start of the period, i.e. a job can only read data that was already available at the start of its period. Output data is made logically available just at the end of the period. Actual job execution may take place anywhere inside the period. Thus, the availability of data is independent from the physical execution of the tasks, as long as a given task set is feasible at all. This approach has two advantages: First, it increases the compositionality of application software. Adding new tasks does not change the communication behaviour of the tasks, as long as the system is still schedulable. Once the parameters for e.g. a software controller have been tuned for the communication schedule defined by LET, they can be kept regardless of additional tasks. Second, with the same argument, the software can easily be ported to other hardware platforms as long as the periods and schedulability can be kept.

An extension of Giotto for distributed system is proposed in [14]. A priori, timing interfaces are defined for tasks that determine the time slots that the tasks can use for computation and communication. The task implementations must adhere to these. For communication between tasks running on different nodes, the underlying network must be able to give timing guarantees such that the predefined time slots can be met. Additionally, a time synchronization between the different nodes is required, which can be achieved through the predictable network. Actual implementations of the tasks can be performed independently from each other.

A more general concept has been defined for OASIS [5]. Here, a task is subdivided into subtasks that are executed sequentially. Each subtask has its own LET. Evolution of the logical time is performed explicitly by an advance instruction `adv(time)` that is called at the end of each subtask.

The Patmos multicore processor with the Argo NoC provides a good hardware platform for the implementation of the LET model. Both enable a time-predictable execution, thus allowing to fix time slots for computation and communication a priori. The Argo NoC can supply a global timing signal that can drive time counters for physical time on each node synchronously. The counters are synchronized by the bootstrap reset sequence of the NoC [18].

To support the LET model, the OS has to provide special communication primitives to the application(s). Traditionally, data sent by a task to another one is available as soon as it has been physically received. LET requires that a task instance can only read data that was already available at the task's activation. This behaviour must be reflected by the communication primitives of the OS.

5.1 Implementation in MOSSCA

In our implementation, we borrow the idea of the `adv` instruction from OASIS. The kernel manages a logical clock that is implemented as a counter variable. Each time the application issues an `adv` call, the counter is advanced and execution of the application is suspended until the physical time matches the logical time.

For communication, a read/write semantic is implemented, i.e. only the most recent value received over a communication channel is relevant, older values can be discarded. The term "most recent" in this context means the value that is available at the start of a subtask's time slot, regardless when the read operation is performed during the time slot. For each communication channel, a cyclic buffer is allocated on the receiver node. A single element in the buffer consists of its logical availability time and the actual data. The size s_B of the buffer depends on the periods of the sender and receiver tasks P_S and P_R. The buffer must be able to hold as many elements as can arrive during one period of the receiver task, plus the element that arrived just before the start of the period. This is to ensure that even if the receiver physically reads the data just at the end of its period, the relevant element is still available. Thus, the buffer size is calculated as:

$$s_B = \left\lceil \frac{P_R}{P_S} \right\rceil + 1 \tag{1}$$

The write locations in the buffer are managed by the sender node using a counter. The send command (see alg. 1) can be issued at any time during the sender's execution. The command copies the data to the communication SPM and marks the message for sending. The actual transmission of the data is deferred until the sender issues an `adv` call (alg. 2), as just then the availability time can be determined. On receiver side, reading from the buffer returns the element that was available at the start of the reader's current time slot, i.e. its current logical time (recall that the LET is advanced just at the end of a job).

Algorithm 1: Send command

```
1 Function send_message(message, dest)
2     select send channel buffer $b_{\text{dest}}$;
3     copy data to $b_{\text{dest}}$;
4     mark $b_{\text{dest}}$ for sending;
```

Algorithm 2: `adv` command

```
1 Function adv(time)
2     ltime ← ltime + time;
3     foreach outgoing channel c do
4         if Message m buffered for sending over c then
5             Setup DMA for transfer of m;
6         end
7     end
8     while physical_time < ltime do
9     end
```

If both sender and receiver thread are executed periodically with periods P_S resp. P_R, then a direct access to the buffer is possible. Instance n of the receiver ($n = 0, 1, \ldots$) reads the value from instance k of the sender, where

$$k = \left\lfloor \frac{nP_R}{P_S} \right\rfloor \tag{2}$$

Then, the buffer offset o_b is:

$$o_b = k \bmod s_B \tag{3}$$

If the more general execution model from OASIS is used, where subsequent segments of tasks can have different LETs, either a more complex calculation is necessary, or the correct element can be found by simply looping through the buffer. The execution time of the loop is bounded by the size of the buffer.

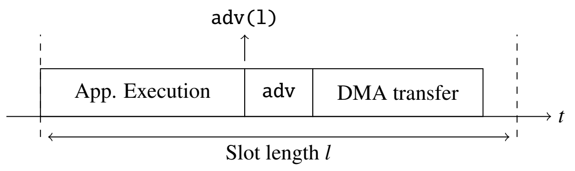

Figure 3: Time slot

5.2 Length of Time Slots

The slot durations, i.e., the parameters of the advance calls, are derived from application requirements. However, they must also take into account the additional work of the `adv` call (see fig. 3). First, the handling of the advance call itself takes some time. During this time, especially the send operations are started. Second, the DMA transfers for the send operations must be finished at the end of the time slot, as then the data must be available at the receiver nodes. Finally, ongoing DMA transfers may slow down the setup of further transfers (line 5 in alg. 2). Therefore, additional time must be provided inside the time slot.

The minimum slot length could be reduced, if the OS knows the length of the slot already at its beginning. Then, the availability time of data can already be determined when a send command is issued. Thus, transmission of data could already be started when the send command is executed, and the DMA transfer could overlap with further computations.

6. QUANTITATIVE PROPERTIES

The following results are based on a small sample application consisting of three nodes that mimic a sensor/computation/actuator system, where every second a message is passed through all three nodes. Each message has a size of 16 bytes.

6.1 Code Size

In the current implementation, the functions for management of LET and LET communication increase the size of the MOSSCA kernel code section by about 2 kB. The whole code section has a size of about 25 kB. As the current implementation is a proof of concept, it is not yet optimized for size. We expect that the size can be reduced strongly through optimizations. For comparison, the text sections of the MOSSCA reference implementation on a ARMV6-M ISA take only around 7.5 kB of memory, with about 2.3 kB for output functions. A part of this large difference between the Patmos and the ARMV6-M code sizes stems from the fact, that the ARMV6-M images mostly uses Thumb instructions, most of which have a width of only 16 Bits, whereas Patmos instructions have at least 32 Bits. Also, the implementation for Patmos uses some additional libraries for NoC access that are not needed in the MOSSCA reference implementation.

Concerning kernel data, only a counter variable must be added. For management of the channels, another integer variable that holds the size of the receive buffer is added to the management structures. As we expect only few channels to be managed on a single node, this memory overhead for data is negligible in our view.

6.2 Execution Time of LET Extension

We have measured the execution times of relevant parts of our extensions. Table 1 shows the numbers for the first implementa-

Table 1: Minimum and maximum execution times of important code parts (kernel executed from global memory)

	min	max
send command	3,201	6,257
recv command	4,276	4,279
adv check	3.171	3,255
adv DMA setup	2,352	2,520

tion of MOSSCA, where kernel code is executed from the global memory. Although minimum and maximum values span a larger interval for some operation, the actual behaviour is quite regular. For the send command (algorithm 1), the extreme values occur only once, while all other instances take 6,173 cycles. Similar behaviour is exhibited by the receive command which loops through receive buffer. Here, most executions take 4,279 cycles. The *adv check* times are calculated from the execution time of the whole loop in the adv function (lines 3-7 in algorithm 1), with the times for setting up the DMA channels (line 5) deducted. In our experiments, the time for the check is constant on each node and depends only on the actual position of the node. The minimum value of 3,171 cycles is found on node 1, while the check on node 3 always takes 3,255 cycles. The execution times for setting up the DMA channels alternate between the shown minimum and maximum values on any node. This is due to the organisation of the actual send buffer, which has provides two places. When both places have been used, some additional management work is performed by the software.

The high execution times stems mostly from cache misses and the necessity for off-chip memory accesses. In the Patmos 4-core, a burst fetch of four 32-bit words from external memory takes in the worst case 104 clock cycles. Thus, even though the functions are rather short, e.g. < 128 instructions for the send call, they have a high execution time.

If kernel code is executed from the ISPM, the execution times decrease drastically. Table 2 shows the measured execution times for the same functions as described above. Aside from the reduction, the execution shows the same regular behaviour as in implementation using global memory for storing kernel code.

The worst-case transmission time of a message is bounded and can be computed as shown in [31].

7. CONCLUSIONS AND OUTLOOK

We have ported the MOSSCA operating system to the Patmos multicore processor and enhanced the implementation by communication mechanisms that adhere to the LET concept. Thus, the compositionality of applications is increased, as changes in a task set no longer influence the other tasks' communication behavior. The LET extensions of MOSSCA use the Argo message passing NoC of the multicore processor. Messages are passed to buffers on the receiver node, and made available to the application only after their logical arrival time. As the Argo NoC uses time-division multiplexing for arbitration, the maximum transfer time of a message can be bounded. Furthermore, the Argo NoC provides a global time base that is used to drive the clock counters of all cores.

Runtime measurements of the LET extensions mainly point out the disadvantages of execution code from a global memory with long access latencies. Even code sequences of only a few hundred instructions take thousands of cycles to execute. We could achieve a drastic reduction of execution times by executing the kernel code from ISPM.

Table 2: Minimum and maximum execution times of important code parts (kernel executed from ISPM)

	min	max
send command	429	675
recv command	751	836
adv check	159	273
adv DMA setup	533	751

The LET extensions to MOSSCA increase the code size by less than 10 %. We expect that we will reduce this overhead in the future through an optimization of the MOSSCA implementation. Thus, we will be able use the SPMs of the single cores more efficiently, and especially make more space available for applications.

8. REFERENCES

[1] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[2] A. Baldovin, E. Mezzetti, and T. Vardanega. A time-composable operating system. In *WCET*, pages 69–80, 2012.

[3] R. L. Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, Jan 1991.

[4] R. L. Cruz. A calculus for network delay. II. Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, Jan 1991.

[5] V. David, J. Delcoigne, E. Leret, A. Ourghanlian, P. Hilsenkopf, and P. Paris. Safety properties ensured by the oasis model for safety critical real-time systems. In W. Ehrenberger, editor, *Computer Safety, Reliability and Security*, volume 1516 of *Lecture Notes in Computer Science*, pages 45–59. Springer Berlin Heidelberg, 1998.

[6] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl. A method cache for Patmos. In *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, pages 100–108, Reno, Nevada, USA, June 2014. IEEE.

[7] J. Delange and L. Lec. Pok, an arinc653-compliant operating system released under the bsd license. *13th Real-Time Linux Workshop*, 10 2011.

[8] B. Dupont de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti. Guaranteed services of the NoC of a manycore processor. In *International Workshop on Network on Chip Architectures (NoCArc)*, pages 11–16, New York, NY, USA, Dec. 2014. ACM.

[9] B. Dupont de Dinechin, D. van Amstel, M. Poulhiès, and G. 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.

[10] K. Goossens and A. Hansson. The AEthereal network on chip after ten years: Goals, evolution, lessons, and future. In *Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC 2010)*, pages 306 –311, 2010.

[11] A. Hansson, M. Subburaman, and K. Goossens. aelite: a flit-synchronous network on chip with composable and

predictable services. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2009)*, pages 250–255, Leuven, Belgium, 2009.

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

[13] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84 – 99, Jan. 2003.

[14] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed giotto. In *2005 ACM SIGPLAN/SIGBED Conference on Languages , Compilers, and Tools for Embedded Systems*, LCTES '05, pages 21–30, New York, NY, USA, 2005. ACM.

[15] B. Huber, S. Hepp, and M. Schoeberl. Scope-based method cache analysis. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 73–82, Madrid, Spain, July 2014.

[16] L. S. Indrusiak. End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration. *Journal of Systems Architecture*, 60(7):553–561, 2014.

[17] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, and J. Sparsø. Argo: A real-time network-on-chip architecture with an efficient GALS implementation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 24:479–492, 2016.

[18] E. Kasapaki and J. Sparsø. Argo: A time-elastic time-division-multiplexed noc using asynchronous routers. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 45–52, May 2014.

[19] L. M. Kinnan. Use of multicore processors in avionics systems and its potential impact on implementation and certification. In *28th IEEE/AIAA Digital Avionics Systems Conference, 2009 (DASC '09)*, pages 1.E.4–1–1.E.4–6, Oct. 2009.

[20] F. Kluge, M. Gerdes, and T. Ungerer. The boot process in real-time manycore processors. In *22nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 77:77–77:86, New York, NY, USA, 2014. ACM.

[21] F. Kluge, M. Gerdes, and T. Ungerer. An operating system for safety-critical applications on manycore processors. In *17th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2014*, pages 238–245. IEEE, June 2014.

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

[23] S. Metzlaff, J. Mische, and T. Ungerer. A Real-Time Capable Many-Core Model. In *Work-in-Progress Session of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*, Vienna, Austria, Nov. 2011.

[24] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Ninth European Dependable Computing Conference, Sibiu, Romania*, pages 132–143. IEEE Computer Society, May 2012.

[25] A. Olofsson, T. Nordström, and Z. ul Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In M. B. Matthews, editor, *ACSSC*, pages 1719–1726. IEEE, 2014.

[26] W. Puffitsch, R. B. Sørensen, and M. Schoeberl. Time-division multiplexing vs network calculus: A comparison. In *Proceedings of the 23th International Conference on Real-Time and Network Systems (RTNS 2015)*, Lille, France, November 2015.

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

[28] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø. A time-predictable memory network-on-chip. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 53–62, Madrid, Spain, July 2014.

[29] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. 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.

[30] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 161–170, April 2008.

[31] R. B. Sørensen, W. Puffitsch, M. Schoeberl, and J. 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.

[32] J. Sparsø, E. Kasapaki, and M. Schoeberl. An area-efficient network interface for a TDM-based network-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1044–1047, San Jose, CA, USA, 2013. EDA Consortium.

[33] T. Ungerer, C. Bradatsch, M. Frieb, F. Kluge, J. Mische, A. Stegmeier, R. Jahr, M. Gerdes, P. Zaykov, L. Matusova, Z. J. J. Li, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, N. Lay, D. George, I. Broster, E. Quiñones, M. Panic, J. Abella, C. Hernandez, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. Parallelizing industrial hard real-time applications for the parMERASA multicore. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(3):53:1–53:27, May 2016.

[34] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of HRT applications supporting analyzability. *IEEE Micro*, 30:66–75, 2010.

[35] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, Apr. 2009.

[36] M. Ziccardi, M. Schoeberl, and T. Vardanega. A time-composable operating system for the Patmos processor. In *The 30th ACM/SIGAPP Symposium On Applied Computing, Embedded Systems Track*, Salamanca, Spain., April 13–17 2015. ACM Press.