A multicore processor for time-critical applications

Schoeberl, Martin; Pezzarossa, Luca; Sparsø, Jens

Published in:
I E E E Design & Test

Link to article, DOI:
10.1109/MDAT.2018.2791809

Publication date:
2018

Document Version
Peer reviewed version

A Multicore Processor for Time-Critical Applications

Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø

Department of Applied Mathematics and Computer Science
Technical University of Denmark, Kgs. Lyngby
Email: {masca, lpez, jspa}@dtu.dk

***Abstract*—Time-critical applications need a processor and software where it is possible to prove that all critical tasks will complete in time. Such a processor is significantly different from a mainstream processor for a notebook, server, or smartphone. It is designed to excel on the worst-case execution time performance and not on the average case performance. This paper presents such a radically different design of a multicore processor for future time-critical systems.**

***Index Terms*—real-time systems, time-predictable computer architecture.**

I. Introduction

Time-critical applications need time-predictable computing platforms to deliver results in time. Imagine, as an example, an airbag system in a car. A crash is detected by several sensors and a computer combines the information from these sensors. The computer decides if the information from the sensors is an indication of a crash and the passengers need some additional protection. The result of this decision may be to tighten the safety belts and to fire the airbags. This must be done in time. If the systems reacts 10 ms too late, it may have catastrophic consequences.

Anti-lock braking systems in a car, automatic aircraft flight control systems, and self-driving cars are further examples where the computer-based control system must respond within a bounded time. Such systems are called *real-time systems*, as stated by Stankovic [10]:

> In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the result is produced.

In this context, the focus is on worst case rather than average case performance. It is important to understand that the worst-case execution time (WCET) of a task executing on a computer cannot be measured easily or quickly. It must be computed by analyzing the system [12]. This is a complex task typically involving simplifications that result in (safe) over-approximations. Such over-approximations represent a loss of performance, and for conventional processors like the one in a laptop this can be as high as one or more orders of magnitude.

Several researchers have pioneered the view that we need a new class of computers that are optimized for WCET and intended for use in real-time systems, safety-critical systems, and cyber-physical systems [3], [8]. It is important to realize that "optimized for the worst case" means more than the processor/computer itself delivering good performance, it is equally important that it is possible—and hopefully even simple—to compute the WCET of a task with minimum over-approximation. This focus involves all layers: the processor hardware, the memory system, the interconnect, the compiler, the operating system, and the application. However, the starting point is a time-predictable hardware platform.

Future real-time systems require more performance than can be delivered by a single core processor. Therefore, we propose a time-predictable multicore processor with time-predictable communication between the cores. One example where more processing power will be needed are future self-driving cars. Another advantage of a platform with many processor cores is that tasks can be assigned their own processor cores.

This paper presents a multicore architecture that is designed and optimized for time-critical systems and supports WCET analysis. All design decisions consider reduction of the WCET instead of the improvement of the average case performance. In our architecture, we use time to trigger and control access to shared resources: memory accesses and message passing. The presented architecture is intended as a platform for safety-critical applications.

Figure 1 shows an overview of our time-predictable multicore processor, called T-CREST [9]. T-CREST consists of processor cores, called Patmos, and two communication structures, both implemented as packed switched networks on chip (NoC). One NoC, called Argo, is used exclusively for inter processor communication (message passing) and it uses a bi-torus topology [5]. The other NoC uses a tree structure and it supports access to a memory controller administering a shared DRAM memory, serving as main memory. Each processor has a small private scratchpad memory (SPM), built out of static memory cells, and messages are passed by writing data from the sender's SPM into the SPM of the receiving core.

The two NoCs and the memory (controller) are shared resources and therefore in principle subject to interference among otherwise unrelated activities. Such interference can severely hurt the WCET of application software executing on the processors. Minimizing or even eliminating interference is obviously highly desirable in a real-time system, and towards this end we have adopted time division multiplexing (TDM) as the underlying principle. This fundamental decision is mo-

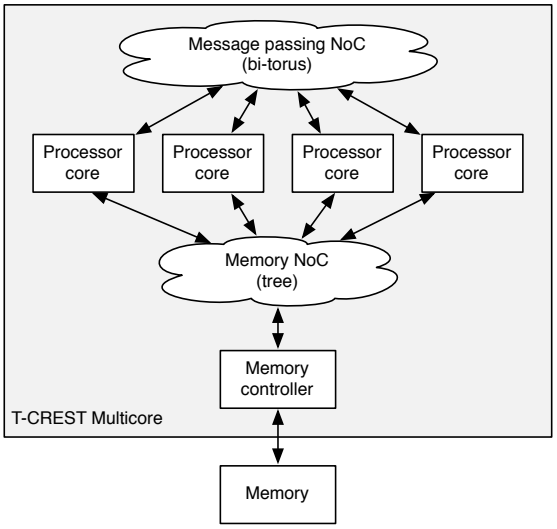

Fig. 1. The T-CREST multicore architecture with several processor cores connected to two NoCs: one for core-to-core message passing and one for access to the shared, external memory

tivated and discussed in more detail in the following section.

Our design is in line with the PRET architecture [3], where Edwards and Lee state: "It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function." Similarly, the Kalray MPPA-256 processor [2] contains 256 processing cores where the cores are designed to be an easy target for WCET analysis. The processing cores are organized in 16 clusters, where the clusters are connected by a NoC that is designed to guarantee bandwidth and latency of messages.

The IDMAC platform [11] uses a similar clustered topology. It is intended for mixed criticality systems. Its NoC is designed to support isolation and aiming to optimize latency and bandwidth it includes a run-time mechanism that adopts to varying and bursty traffic.

The CoMPSoC platform [4] removes all application interferences by resource reservation. CoMPSoC combines the AEthereal style NoC with customized processor cores from Silicon Hive and a composable memory controller. In contrast to the T-CREST platform, no caches are supported and all code needs to fit into on-chip memory. Our NoC architecture was inspired by the family of AEthereal NoCs, where the packets follow a static, time-driven schedule.

The Reduced Complexity Many-Core architecture [7] proposes to avoid shared memory at all and to support timing analysis by using a fine-grain message passing NoC. We agree on this approach to prefer the Argo message passing NoC over shared memory communication, but also allow shared external memory for larger code and larger data structures.

II. Time Division Multiplexing

Using time as an arbitration mechanism is a well-known scheme and is known as TDM or as time-triggered architecture [6]. The main benefit is that arbitration decisions are

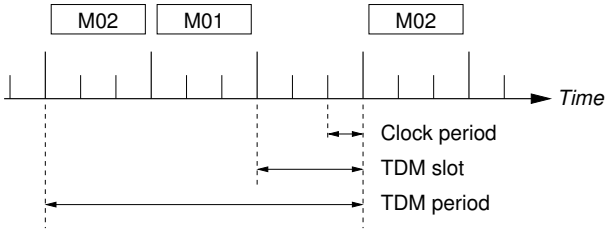

Fig. 2. An example TDM schedule with a period of 3 slots.

calculated offline, and captured in a static schedule: time is divided into slots. Each slot represents a point in time where exactly one client can access the resource. The mapping of clients to time slots is prescribed by that static schedule.

In the Argo message passing NoC the resources are the individual links that connect routers, and it is possible and desirable to compute a global schedule that allow a packet that need to traverse a sequence of links to traverse these in pipelined manner. Figure 2 shows an example schedule for processor P_0 in a small system with four processors P_0, P_1, P_2, P_3. The period of the schedule is 3 slots and each slot is 3 clock cycles. In the first slot P_0 sends a packet (M02) to P_2 and in the second slot it sends a packet (M01) to $P1$. The third slot is not used by P_0. After the TDM schedule is finished a new one starts with another packet to P_2.

The use of such static schedules has two main advantages: (1) it is time predictable, as the timing and the time to wait for an access slot can be bounded offline and (2) the hardware for enforcing the static schedule is simple and scales well. When all clients share a common notion of time, it is possible to distribute the schedule and make local arbitration decisions (from which the global and pre-computed schedule will emerge).

Individual messages fit exactly into a TDM slot. Therefore, there is no need to preempt any message. When a message is sent within its slot it moves through the NoC without any blocking or preemption till the destination. If flow control between tasks is needed, it is performed in software.

One known downside of TDM based arbitration is that this scheme is not what is called "work conserving": if a client has reserved a slot but does not use the resource, then the resource is not offered to other clients that could use it. It is our experience that some consider this as "waste of resources". When focus is on WCET we consider this view a fallacy, as meeting deadlines is all that matters.

The alternative is to implement mechanisms that make arbitration decisions on the fly during runtime. This is the approach taken by most designers, but it incurs a considerable hardware cost. In a NoC, run-time arbitration brings with it a need to buffer temporarily stalled data and the hardware implementation of most NoCs is dominated by buffers—possibly several per router port if virtual circuits are used. On top of this comes the arbitration circuitry itself, flow control among buffers, and in some designs, large crossbars that switch among individual virtual circuits in each router

port.

In contrast to this, a router for a TDM-based NoC is merely a pipelined X-Y switch, and the area of a TDM router is typically a small fraction of that of a typical router for a work conserving NoC. This means that for the same hardware cost, a TDM NoC can provide substantially higher bandwidth. This invalidates or at least makes doubtful the objections against TDM networks for not being work conserving.

In conclusion, the use of time division multiplexing is not only a means to achieve time predictability and guarantee WCET, it is also a choice that results in efficient hardware implementations. TDM is a recurrent theme in our architecture and we use it at several levels:

- at the Argo message passing NoC,
- in the NI for the Argo message passing NoC,
- at the memory NoC,
- and for DRAM refresh.

III. The Patmos Processor

Patmos, the processing core of the multicore processor, was designed to be time-predictable. All features are optimized primarily to reduce the WCET bound. We added only features where WCET analysis is possible. Patmos contains a dual-issue in-order pipeline. The compiler statically schedules multiple instructions that can execute in parallel. The pipeline is organized so that there are no timing dependencies between instructions. For example, all cache misses happen in the same pipeline stage (the memory stage). This independency of instruction timings simplifies low-level WCET analysis and furthermore is a guarantee that there are no timing anomalies.

Patmos contains specialized caches for instructions and data: (1) the method cache (M$) and (2) the stack cache (S$). The method cache stores whole methods (or functions in C). This feature simplifies cache analysis, as cache misses can only happen on function call or function return. All instruction fetches within a function are guaranteed cache hits. The compiler is responsible for splitting large functions into smaller ones, if needed. The stack cache is responsible for caching stack allocated data. Addresses of stack allocated data are relatively easy to deal with in the WCET analysis. Therefore, we provide an extra cache for stack allocated data, the stack cache. The compiler emits instructions on function entry and function return to ensure that the stack cache is valid. That means that all stack access instructions within a function are guaranteed cache hits. The data cache (D$) caches static and heap allocated data. For program-managed code and data, Patmos supports instruction and data SPMs. The data SPM is also used as memory for the Argo message passing NoC. Figure 3(a) shows Patmos with its caches and the SPM and the connection to the Argo message passing NoC.

Another feature of Patmos is its support of the single-path code paradigm. With single-path code, all data dependent control flow decisions are transformed to a single execution path and conditional update of processor state. When all data dependent timing variations are removed, the execution time of a task becomes constant. Therefore, a simple measurement is a valid approach to finding the WCET. Single-path code needs, as WCET analyzable code in general, known upper bounds on loops. Patmos supports the single-path code generation with instructions that can, depending on a predicate, be conditionally executed.

Patmos is supported by a port of the LLVM compiler. The compiler also supports the special features of Patmos, e.g., generation of instructions for the stack cache, function splitting for the method cache, or generation of single-path code.

The simplicity of the processor and the timing independence of instructions directly translates into simpler WCET analysis. E.g., no pipeline simulation, with its possible state explosion, is needed during WCET analysis. Up to now, two WCET analysis tools support Patmos: aiT[1] and platin. aiT is the standard industrial static WCET analysis tool. Support for Patmos was added during the T-CREST project. The open source tool platin is a research WCET analysis tool, which is part of the compiler distribution for Patmos. Both tools support analyses of the method cache and the stack cache.

IV. The Argo Message Passing NoC

Message passing involves copying data from local memory in the sender across a "channel" and into local memory in the receiver. The Argo message passing NoC implements this functionality using TDM and static scheduling, which effectively creates private end-to-end channels. This approach avoids interference between independent traffic flows and it makes it possible to calculate bounds on the worst-case latency of sending a message.

Similar TDM based message passing functionality is seen in other multicore platforms; what makes our NoC different is its efficient and low cost network interface microarchitecture that avoids resources for arbitration, buffering, and flow control. In addition, if using asynchronous router implementations, our NoC supports different forms of relaxed synchrony across the entire platform without the use of synchronization FIFOs. Details on both topics are published in [5]. Below we explain the key features.

The Argo message passing NoC is composed of a packet switched structure of routers and links in a bi-torus topology. By using TDM in combination with source routing the routers are merely pipelined X-Y switches controlled by route information in the packet headers. Each processor is connected to a router using a network interface (NI), as seen in Figure 3, and it is the NIs that implement message passing.

Each NI operates in a synchronous fashion and executes a fraction of the global schedule by sourcing data into the channels that originate from its core. As an analogy, we mention that this is like the list of departures announced at a railway station, and continuing this analogy we observe that if all wall clocks on all railway stations are synchronized, if all trains depart according to schedule and travel at the prescribed speed, then no train will ever encounter another train on the same track (and hence no signaling lights are needed). In

[1] https://www.absint.com/ait/

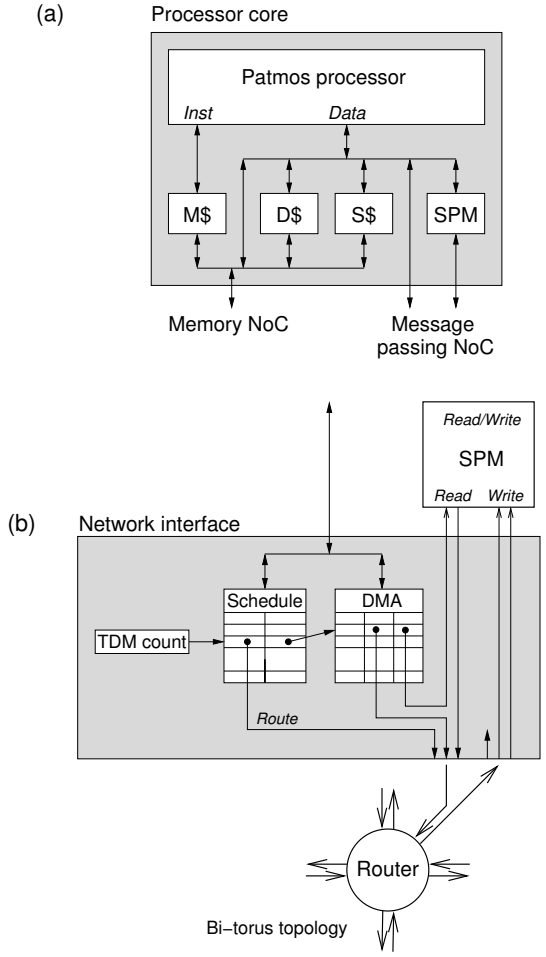

Fig. 3. Details of a Patmos processor node and the network interface of the Argo message passing NoC

hardware, this corresponds to arbitration and buffering not being needed.

In a platform with N cores there are typically $N-1$ outgoing channels allowing the core to send data to all the other cores. Each of these is driven by a corresponding DMA controller. If multiple tasks share a processor, some outgoing channels from that processor may have to be shared as well. Alternatively, the NI must be set up to provide multiple channels between pairs of processors, such that tasks can communicate using non-shared channels. The architecture supports this as well.

In a simple and straightforward implementation of the NoC, packets consist of three 32-bit words sent in sequence and a slot in the TDM schedule is also 3 clock cycles. With 64 cores a typical TDM schedule is 50–100 slots, corresponding to 150–300 clock cycles. This means that long messages are sent as a sequence of small packets and that messages originating from one core are sent in an interleaved fashion. It also means that the time to transmit a long message is determined by the throughput of the channel rather than by the latency experienced by a single packet. In some applications, it will be relevant to assign multiple slots in a TDM period to a certain channel. The network uses shortest path routing and packets belonging to a message are guaranteed to arrive in order.

Our NI microarchitecture, shown in Figure 3(b), integrates DMA controllers and TDM scheduling in a novel way and data is transferred end-to-end from the source SPM and into the destination SPM, controlled by the TDM schedule. Because of that, our design enjoys the properties of TDM: it avoids all buffering and flow control and it is 2–4 times smaller than existing designs with similar functionality [5].

The TDM counter corresponds to the wall clock in the railway analogy and it repeatedly indexes through the slots in the TDM schedule. For every allocated slot in the schedule the NI may send a packet. If a packet is to be sent the indexed schedule table entry provides the route for the packet and a pointer to the DMA controller that will source the payload data and the target address of the packet. This is illustrated in Figure 3(b) where the entry in the schedule table points to an entry in the DMA table. As only one DMA controller can be active at any given time, the read pointer into the source SPM, the write pointer into destination SPM, and the word count are stored as a record in the DMA table. As illustrated, the read pointer is used to read from the local SPM and the outgoing packet is assembled from the route, the write pointer, and the payload data that is read from the SPM at exactly the time it is transmitted. When a packet arrives at a NI the payload data is written directly into the target SPM at the address carried in the packet header. Note that this does not involve a DMA controller or any other form of control.

The mechanism described above naturally lends itself to a globally synchronous implementation. As outlined in [5] implementations with more relaxed timing organizations are possible at no or very little extra hardware. In the most extreme case the NIs can operate with an unknown upwards bounded (clock) skew of 1–2 cycles and the processors can operate with completely independent clocks.

V. Shared Memory NoC and DRAM Memory Interface

For realistically sized applications and larger data structures external memory is used. Access to this shared memory needs to be time-predictable as well to bound access times for loads, stores, and cache fills or spills.

For the shared, external memory we use a dedicated memory NoC. As access to this memory is a many-to-one relationship between processor cores and the memory, we organize this NoC as a tree with channels towards the memory controller and a return path for read data. Figure 4 shows the organization of the memory NoC. Each processor core is connected to a network interface (NI). The NIs are connected by a tree of merge circuits downstream towards the memory interface (MI) and back upstream for the return data. The MI is connected to the on-chip memory controller, which itself is connected to the external memory. The interface between the processor and the NI is the same interface between the MI and the memory controller.

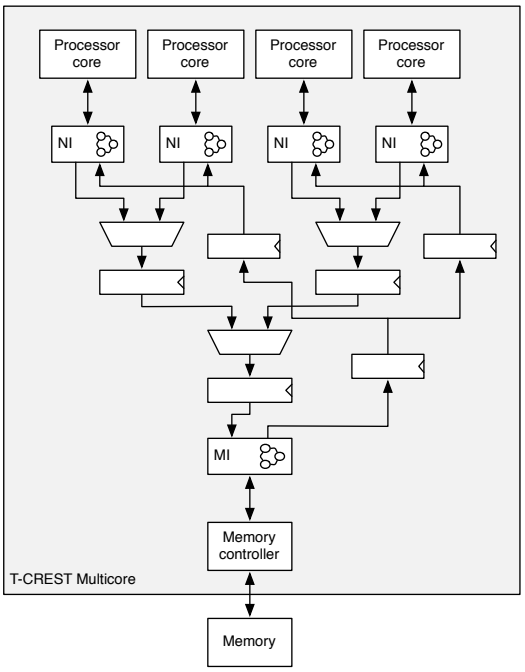

Fig. 4. The distributed TDM based memory NoC

We organize the TDM based arbitration in a distributed manner. Each core's local NI executes the common TDM schedule. When the time slot for the core arrives, and a memory transaction is pending, the NI acknowledges the transaction to the processor core and the transaction freely flows down the network tree. No flow control, arbitration, or buffering (except pipeline registers to improve clock frequency) is performed along the downstream path. The memory request arrives at the MI and is forwarded to the memory controller.

On a read transaction, the result is sent from the memory back upstream to the processor cores. The NI knows exactly when the read data for its read request shall be returned. The NI uses time to filter out read data targeting other cores.

The worst-case timing of memory accesses (M$ filling and access to shared data) is dependent on the number of cores (number of TDM slots). Section VII shows how to compute the worst-case memory timing, which itself is included in the WCET analysis.

To enable TDM based scheduling of memory accesses the worst-case memory access time needs to be known. Standard DRAM controllers optimize for average case performance by reordering requests and keeping pages open. However, for accesses to the memory the worst case is when accesses go to different rows. We optimize for the WCET and use a closed page policy, as this results in the shortest worst case latency [1].

DRAM memory needs regular refresh, during which the memory cannot be accessed. In the case of general purpose computing this refresh is just a small reduction of performance. However, for real-time systems where the WCET is of primary importance we also need to consider the clock cycles where the memory is not available due to refresh.

When the external memory is a DRAM device that needs refresh, a refresh circuit is added to the memory NoC at the same level as a processor core. Therefore, refresh consumes one TDM slot, but has no further influence on the memory access timing.

Refresh can be considered in real-time systems in two ways: (1) model the refresh as a periodic task and (2) include the possible refresh in the memory access time. For a single core processor, where multiple periodic tasks already need to be considered in the schedulability analysis, the additional task is easy to consider. Typical values for this task are a period of 7.8 μs and a WCET of 60 ns.

However, with a multicore processor the memory is already shared with a TDM based arbiter. Therefore, it is simpler to add a short TDM slot for the refresh into the TDM schedule. For the WCET analysis of memory access, i.e., cache misses, the TDM schedule is already considered. We just increase the worst-case waiting time for the memory access by the additional refresh slot.

VI. Programming the Multicore

The T-CREST platform provides the hardware infrastructure to build time-predictable applications, organized as a set of communicating tasks executing on a multicore. However, there is no common agreed approach how to structure real-time applications on multicore processors. Therefore, we aim to support several different programming paradigms to support future research.

For task scheduling we provide following support:

- The Patmos processor supports timer interrupts and therefore classic preemptive real-time operating systems can execute on T-CREST. For example, RTEMS has been ported to T-CREST.
- Due to the TDM approach in the Argo message passing NoC all cores have a common notion of time, even when connected via an asynchronous version of the NoC. Therefore, T-CREST also supports time-triggered execution of tasks, where time can be used to enforce precedence constraints.
- The NoC also supports cross-core interrupts, which enables implementation of a multicore global scheduler. Therefore, any multicore real-time scheduling algorithm can be explored on T-CREST.

For communication between tasks we provide following support:

- Mainstream communication between threads (or tasks) is via data structures allocated in shared memory and protected by locks. Although, this communication does not scale very well on multicore processors, T-CREST supports this form of communication.
- An alternative form of communication between threads/tasks is message passing. It can be implemented via shared memory, which T-CREST supports with time-predictable access to the shared main memory. However,

Component	Hardware (LC)	Memory
Shared resources:		
Memory controller	243	0.0 KB
Memory NoC	2316	0.0 KB
Per core:		
Patmos	4353	0.5 KB
NoC NI	761	1.3 KB
NoC router	686	0.0 KB
Memory per core:		
Boot ROM	0	0.5 KB
M$	1558	8.0 KB
D$	641	4.6 KB
S$	959	4.0 KB
SPM	0	4.0 KB
Total (9 cores)	85706	212.2 KB

TABLE I
RESOURCE CONSUMPTIONS OF DIFFERENT COMPONENTS OF A 9 CORE T-CREST PLATFORM

it is more efficient (higher bandwidth, shorter latency, and less energy consumption) when the messages are passed between cores without leaving the chip. The Argo message passing NoC supports this with transfer of messages between core local SPMs.

We just started to explore different possibilities to organize applications into communicating tasks on top of T-CREST. The T-CREST platform has been evaluated with an avionics use case [9]. Furthermore, a DSP audio effect application has been developed for T-CREST that uses multiple cores to implement a signal processing pipeline. Future work is in progress to use T-CREST in an advanced Drone platform[2] and as a Fog node for robotics and industrial automation.[3]

We have no final answer what is the best solution for programming a multicore architecture. Probably the best solution depends on the type of application. We consider this as exciting future work to explore, and especially compare different execution models for a multicore architecture on the same platform.

VII. EVALUATION AND DISCUSSION

For the evaluation, we use a 9-core prototype implemented using the popular and inexpensive Altera DE2-115 board with a Cyclone IV EP4CE115 FPGA. The Argo message passing NoC is configured with an all-to-all schedule. We synthesized the design with Quartus Prime 16.1 Lite Edition with default settings.

Table I shows the resource consumption of the individual components and the total numbers for a 9-core platform. The resource consumption is given in logic cells (LC) and on-chip memory consumption. One LC in the Cyclone FPGA contains a 4-bit lookup table and a flip-flop. The full design consumes 86 kLCs and 212 KB of memory. Memory consumption is stated separately as the sizes of the memories are configurable.

[2]http://predict.compute.dtu.dk/
[3]http://www.fora-etn.eu/

Calculated per processor, we see that both NoCs are relatively small; 6 % for the memory NoC and 33 % for the Argo message passing NoC. The latter is substantially smaller than other NoC implementations [5].

For both networks, we can derive the worst-case time it takes to convey a single transaction across the network as

$$T_{trans} = T_{wait} + T_{rw} + T_{NoC} \tag{1}$$

where T_{wait} is the *worst case* waiting time for an assigned TDM slot, T_{rw} is the time for a read or write action at the source, and T_{NoC} is the time spent to traverse the NoC. T_{wait} depends on the TDM schedule. The worst case is when a request just missed its own slot by a single clock cycle, which is shown in the first parenthesis in equations 2 and 3 below. T_{rw} and T_{NoC} are constant for a given platform.

For the memory NoC a read or write involves transmitting a single packet in a single slot, whereas in the Argo message passing NoC a write action typically involves transmitting several packets in a sequence of slots.

For the two networks, we get the following transaction latencies:

$$T_{mem} = ((N \cdot s - 1 + r) + s + L_{NoC}) \cdot t_{clk} \tag{2}$$

$$T_{msg} = ((N \cdot s - 1) + (\left\lceil \frac{S_{msg}}{S_{chan}} \right\rceil - 1) \cdot N \cdot s + s) + L_{\mathrm{NoC}}) \cdot t_{clk} \tag{3}$$

where N is the number of TDM slots in the schedule, s the length of a slot expressed in clock cycles, L_{NoC} is the number of clock cycles it takes to traverse the NoC and t_{clk} the clock period. For the memory NoC r is the length of the additional refresh slot. For the Argo message passing NoC S_{msg} is the size of the message (in bytes) and S_{chan} is the number of bytes sent across the channel in one TDM period.

For the memory NoC in our 9-core platform we have: $N = 9$, $s = 10$, $r = 4$ and $L_{NoC} = 3$, resulting in a worst-case time for a 4-word burst of 106 clock cycles.

For the Argo message passing NoC in our 9-core platform a schedule that supports communication among all processors is $N = 10$ slots long. With $s = 3$ clock cycles the TDM period is 30 clock cycles. During this time interval each of the $9 \cdot (9-1) = 72$ channels can transmit $S_{chan} = 8$ bytes, and for long block transfers the NoC can sustain this bandwidth. If all channels are fully utilized each individual processor must produce *and* consume 64 bytes in 30 clock cycles or 2.13 bytes per clock period. This is clearly much more than the available computation power of a core.

The worst case end-to-end latency for sending a message is the WCET of the software function implementing the send primitive (setting up the DMA transfer) plus the time it takes to transfer the message across the Argo message passing NoC as discussed in the previous paragraph. The former can be computed by using a WCET analysis tool such as AbsInt's aiT and the latter is computed using equation 3.

For the 9-core platform using the all-to-all schedule Table II shows the latency for sending messages of different sizes.

Message size (bytes)	8	32	128	512
Latency (cycles)	41	131	491	1931

TABLE II
LATENCIES OF MESSAGES OF DIFFERENT SIZES

If all processors constantly sends 512-byte messages to all other processors each processor will have to consume and produce $8 \cdot (512/1931) = 2.12$ bytes per clock cycle, which is very close to the 2.13 computed above for the Argo message passing NoC alone and much more than the available computation power of a core.

All the above results relate to a 9-core platform. When the number of processors is increased, the bandwidth per processor obviously degrades. For the memory NoC this degradation is linear in the number of processors. For the Argo message passing NoC we note that for platforms with $8 \times 8 = 64$, $10 \times 10 = 100$ and $15 \times 15 = 225$ cores we have computed all-to-all schedules with periods of 85, 157 and 471 slots respectively. With these schedules the bandwidth is 1.98, 1.75 and 0.95 bytes per processor per cycle—even in the latter case still more than the available computation power of a core. This is a strong hint on the scalability and viability of a TDM based NoC. Furthermore, application specific schedules generated for more sparse core communication graphs can result in far better numbers.

Finally, we mention that the performance and scalability of the T-CREST platform is studied in [9] where it is shown that the the T-CREST platform scales better when using multiple cores than the LEON 4 multicore processor.

The time-critical multicore and the tools (compiler and platin WCET analyzer) are available in open source hosted at GitHub: https://github.com/t-crest. The build process on a Linux computer is briefly described in the README and in more detail in the Patmos handbook, available from: https://github.com/t-crest/patmos

VIII. CONCLUSION

Time-critical systems need to guarantee to deliver results in time, therefore they are also called real-time systems. A real-time application consists of real-time tasks that must be written to enable worst-case execution time analysis. However, worst-case execution time of tasks can only be ensured when these tasks are executed on a time-predictable platform.

In this paper, we presented a processor, called T-CREST, consisting of several processor cores, which are optimized for time-predictable computation. These cores simplify worst-case execution time analysis of tasks. The processing cores are connected by two communication structures: a core-to-core messages passing network-on-chip (NoC) and a core-to-memory NoC. Both NoCs use time division multiplexing as an arbitration scheme to allow time-predictable communication.

By avoiding hard to analyze processor features and using time division multiplexing for the NoCs the presented platform is reasonably small. We can prototype a 9-core system on a cheap, medium sized FPGA.

Acknowledgment

We would like to thank all T-CREST team members and students who helped to build this platform and for all the joy of the discussions during the project meetings and dinners: Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Florian Brandner, David VH Chong, Philipp Degasperi, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Edgar Lakis, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Torur Biskopsto Strom, Rasmus Bo Sørensen, Alessandro Tocchi, and Jack Whitham.

REFERENCES

[1] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable sdram memory controller. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, New York, NY, USA, 2007. ACM.

[2] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume 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.

[3] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *Proc. 44th Design Automation Conference (DAC 2007)*, pages 264–265, New York, NY, USA, 2007. ACM.

[4] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, January 2009.

[5] 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(2):479–492, 2016.

[6] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[7] Jörg Mische, Martin Frieb, Alexander Stegmeier, and Theo Ungerer. Reduced complexity many-core: Timing predictability due to message-passing. In Jens Knoop, Wolfgang Karl, Martin Schulz, Koji Inoue, and Thilo Pionteck, editors, *Architecture of Computing Systems - ARCS 2017: 30th International Conference, Vienna, Austria, April 3–6, 2017, Proceedings*, pages 139–151, Cham, 2017. Springer International Publishing.

[8] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

[9] 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.

[10] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[11] Sebastian Tobuschat, Philip Axer, Rolf Ernst, and Jonas Diemer. IDAMC: A NoC for Mixed Criticality Systems. In *Proc. IEEE International Conference on Embedded and Real-time Computing Systems and Applications (RTCSA)*, pages 149–156, 2013.

[12] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

Martin Schoeberl is an Associate Professor in the Department of Applied Mathematics and Computer Science, Technical University of Denmark. His research interests include hard real-time systems, time-predictable computer architecture, and real-time Java. He received a PhD and a Habilitation in computer engineering from the Vienna University of Technology. He is a Member of the IEEE and the ACM.

Luca Pezzarossa is a PhD student in the Department of Applied Mathematics and Computer Science, Technical University of Denmark. His research interests include reconfiguration, real-time systems, embedded applications, networks-on-chip, and system-on-chip design. He received a MSc in electronic engineering from the Marche Polytechnic University, Italy. He is a member of the IEEE.

Jens Sparsø is a Professor in the Department of Applied Mathematics and Computer Science, Technical University of Denmark. His research interests include: digital circuits and systems, asynchronous circuits, many-core architectures and networks-on-chip—in short, hardware platforms for embedded and cyber-physical systems. He is a member of the IEEE.

Contact Information:

Martin Schoeberl
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Richard Petersens Plads
Building 322, room 128
2800 Lyngby
Denmark

Phone +45 45253743
Email masca@dtu.dk