T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 4.1 Reconfigurable memory controller concepts

Version 1.0
14 March 2012
Final

Public Distribution

University of York, Eindhoven University of Technology

Project Partners: **AbsInt Angewandte Informatik**, **Eindhoven University of Technology**, **GMVIS Skysoft**, **Intecs**, **Technical University of Denmark**, **The Open Group**, **University of York**, **Vienna University of Technology**

Project Partner Contact Information

AbsInt Angewandte Informatik Christian Ferdinand Science Park 1 66123 Saarbrücken, Germany Tel: +49 681 383600 Fax: +49 681 3836020 E-mail: ferdinand@absint.com	**Eindhoven University of Technology** Kees Goossens Potentiaal PT 9.34 Den Dolech 2 5612 AZ Eindhoven, The Netherlands E-mail: k.g.w.goossens@tue.nl
GMVIS Skysoft Tobias Schoofs Av. D. Joao II, Torre Fernao Magalhaes, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: tobias.schoofs@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail:
Technical University of Denmark Martin Schoeberl Richard Petersens Plads 2800 Lyngby, Denmark Tel: +45 45 25 37 43 Fax: +45 45 93 00 74 E-mail: masca@imm.dtu.dk	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail: s.hansen@opengroup.org
University of York Neil Audsley Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325 500 E-mail: Neil.Audsley@cs.york.ac.uk	**Vienna University of Technology** Peter Puschner Treitlstrasse 3 1040 Vienna, Austria Tel: +43 1 58801 18227 Fax: +43 1 58801 918227 E-mail: peter@vmars.tuwien.ac.at

Contents

Document Control

Version	Status	Date
0.1	First draft	6 January 2012
0.1	Second draft	1 March 2012
1.0	Final version	14 March 2012

Executive Summary

WP4 (Memory Hierarchy) has the main objective of specification, design and implementation of a *reconfigurable memory hierarchy* that provides predictable performance. Thi lies within the overall goal of T-CREST, which is to provide a *time-predictable computer architecture* for real-time systems. The prime assessment criteria for the computer architecture is its inherent time predictability and the degree of time predictability that can be achieved by applications targeted at the architecture.

This document forms the main deliverable for Task 4.1 (Reconfigurable Real-Time Memory Architecture Specification). It discusses the proposed conceptual aspects of the *memory hierarchy* within T-CREST, and to understand how they contribute to meeting the time-predictability requirements of the architecture.

The reconfigurable T-CREST memory hierarchy will be constructed to facilitate bounded end-to-end timing at the system level. Given that the memory hierarchy is physically fixed within the overall NoC architecture and bus connections to off-chip SDRAM, reconfigurability refers to two particular areas of the memory hierarchy discussed in detail in this document:

1. Dynamic Memory Controller – to increase the flexibility of the main (off-chip) memory, T-CREST aims to develop a dynamically scheduled SDRAM controller .

2. Scratchpad Memory – the inclusion of scratchpad memory within each NoC CPU tile allows software level control of a distributed scratchpad hierarchy whereby data can be moved between any pair of scratchpads in the NoC.

The two aspects of dynamic SDRAM controller for off-chip memory and flexible scratchpad memory within the NoC together constitute the T-CREST reconfigurable memory architecture.

1 Introduction

The goal of T-CREST is to provide a time-predictable computer architecture for real-time systems. The prime assessment criteria for the architecture is its inherent time predictability and the degree of time predictability that can be achieved by applications targeted at the architecture. The purpose of this document is to discuss the proposed conceptual aspects of the *memory hierarchy* within T-CREST, and to understand how they contribute to meeting the time-predictability requirements of the architecture.

The primary measure of time predictability is whether tight accurate end-to-end timing bounds can be established for applications executing on the architecture. This involves considering the fundamental computer architecture (i.e. processor, memory hierarchy, NoC), code generated for the execution (i.e. Compiler), and the programming model used by the application. This enables the timing analysis of the resultant system (i.e. including Worst-Case Execution Time Analysis (WCET)). Given the maximum execution times of individual software components, it can be determined whether timing deadlines involving those components are met. Thus, maximum system end-to-end response times are established and bounded prior to the system being deployed. Conventionally, for safety-critical systems such as aeroplanes or car braking systems, establishing worst-case end-to-end timing behaviour of the system is an essential part of system development, implementation and verification.

The remainder of this section provides further background, with sections 2 and 3 discussing the concepts involved with the on-chip and off-chip aspects of the memory hierarchy respectively.

1.1 Bounding End-to-End Timing in Memory Hierarchies

The effect of the memory hierarchy within the overall timing behaviour of the system has changed greatly as architectures have developed in recent decades. The basic requirement, that of bounding the worst-case end-to-end timing of a system, has not changed. Essentially, the general case involves bounding the time between an input arriving at the system, performing required computation, and producing the appropriate output from the system. This is illustrated in Figure 1(a), noting that the computation can be broken into a number of separate software processes (shown in Figure 1(b)).

The *timing variability* that is seen in the system is due to the variability in executing the software process on the CPU. Three aspects are apparent:

1. *Instructions*: Variation is due to the data and control flow within code. If certain restrictions are made, this can be bounded such that a WCET can be determined.

2. *Memory Hierarchy*: Variation is due to the time taken to access the memory by the CPU, which depends upon the connectivity between CPU and memory and contention experienced.

3. *Scheduling*: Variation is due to the ordering of execution of processes on CPUs, dictated by the scheduling policy. Given WCET of processes and bounded behaviour of the memory hierarchy, and restrictions upon the scheduling policies assumed (e.g. fixed priority, TDM), worst-case end-to-end timing can be determined.

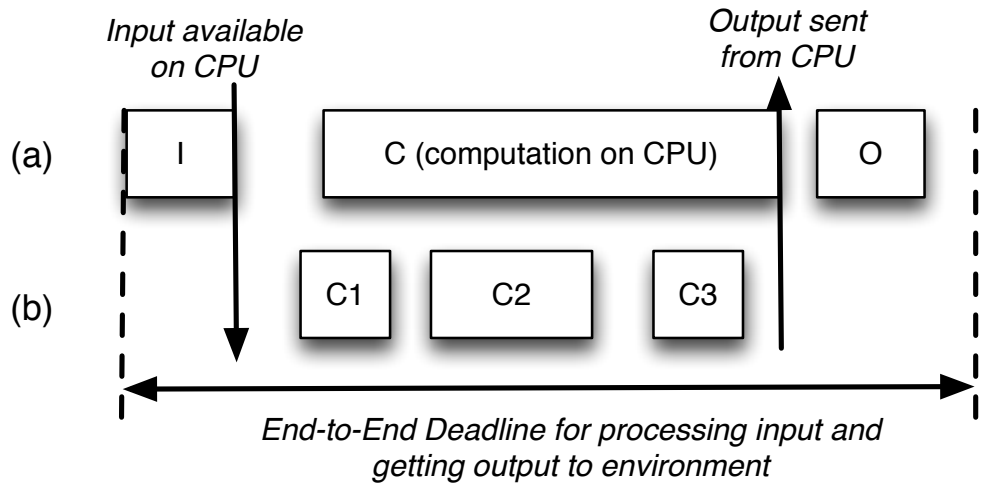

Figure 1: Simple End-to-End Timing Illustrated for Single Input, Computation and Output (a), and for a Chain of Processes (b)

Note: T-CREST will not develop processor scheduling techniques – but remains aware of the issues of scheduling and schedulability analysis, as this is the way that overall system end-to-end timing deadlines can be assessed.

1.2 Timing Variability and Memory Hierarchy Architectures

The role of the memory hierarchy in end-to-end timing becomes increasingly complex as architecture complexity increases:

- *Simplistic uniprocessor architectures without cache*:
  Calculating the effect of the memory hierarchy on application timing is easy on such architectures. The only consideration is the RAM access times (commonly one or two cycles). Accurate WCETs may be determined.
- *Uniprocessor with cache*:
  Caches are introduced as a solution to the discrepancy between CPU speeds and memory speeds. Caches require the development of models within WCET analysis to determine what code and data are within the cache at any time, to determine the worst-case timing of code. Cache misses are likely to cost at least an order of magnitude more (in cycle delays) than accessing the memory in a simple system.
- *Uniprocessor with multi-level cache*:
  Multi-level caches increase the complexity, and the miss penalty.
- *Multiprocessor with multi-level cache, one level shared*:
  For multi-CPU architectures with shared cache, a further issue arises – the cache of one CPU can be affected by the activity of another CPU. To determine WCETs, these effects must be modelled.
- *Network-on-Chip*:
  For NoC architectures, such as assumed within T-CREST, off-chip memory is shared by all

CPUs within the NoC. Hence there is also now the additional problem of understanding the worst-case time to access the network and the shared memory.

Traditionally, WCET analyses have directly included all memory access times within the WCET. Thus, for simple uniprocessor architectures the access times to RAM are easily included within the WCET together with the instruction costs. This is illustrated in Figure 2(a), where some instructions (I) require no memory access; others need memory access (M), in the form of low cycle RAM access.

For cache architectures there is increased timing variability, as memory access times can be either cache hits or misses. Both these effects are modelled and included within the WCET. This is illustrated in Figure 2(b), where one memory access is a cache miss, requiring additional time to bring the data from memory into the cache (M). Variability in timing lies in the difficulty in modelling what is present in cache at any time – assuming that the cache is insufficient to hold all code / data required by the process. This approach is not easily scaled to shared caches, as the effects of other process executions on different processors may have to be modelled to accurately bound the WCET.

These ideas can be extended to NoC architectures, where the delays experienced accessing the NoC can be included within the WCET (Figure 2(c)). However, some memory accesses can be asynchronous, especially when using the NoC (due to the potentially long delays). This is illustrated in Figure 2(d) where a asynchronous scratchpad read is instigated. Clearly some execution may well be delayed by the time the scratchpad read takes, as illustrated.

To ensure that the memory hierarchy within T-CREST and the memory hierarchy concepts outlined in this document are amenable to performing end-to-end timing analysis on the system and individual application processes, we separate the distinct parts of the memory from a timing perspective. Initially, memory within a NoC tile can be considered local memory, where the access times are minimal (few cycles). All other memory accesses require access to the NoC infrastructure to reach the physical site of the required memory (e.g. off-chip SDRAM or on-chip scratchpad memory (SPM) in a different NoC tile). This access time to the remote site must be bounded. The time to perform the access at the memory (e.g. time for the SDRAM off-chip memory controller to respond) must be bounded.

To provide end-to-end timing bounds, the characteristics of the accesses by individual tiles (specifically, the processes running on the CPUs within the tiles) must be sufficiently understood and bounded. This enables the load on the shared communications fabric (i.e. the NoC communications infrastructure) to be bounded – hence end-to-end timing bounds calculated. Bounding the load can be achieved in different ways, from strict computational models (e.g. actor-oriented, where use of the memory-hierarchy across the NoC will be confined to the start and end of a process execution), to TDMA scheduling of the NoC such that a given bandwidth is guaranteed for each NoC CPU or process.

1.3 Reconfigurable Memory Controller Architecture

The T-CREST memory hierarchy will be constructed to facilitate bounded end-to-end timing at the system level. One key concept within T-CREST is the provision of a reconfigurable memory hierarchy. Given that the memory hierarchy is physically fixed within the overall NoC architecture and

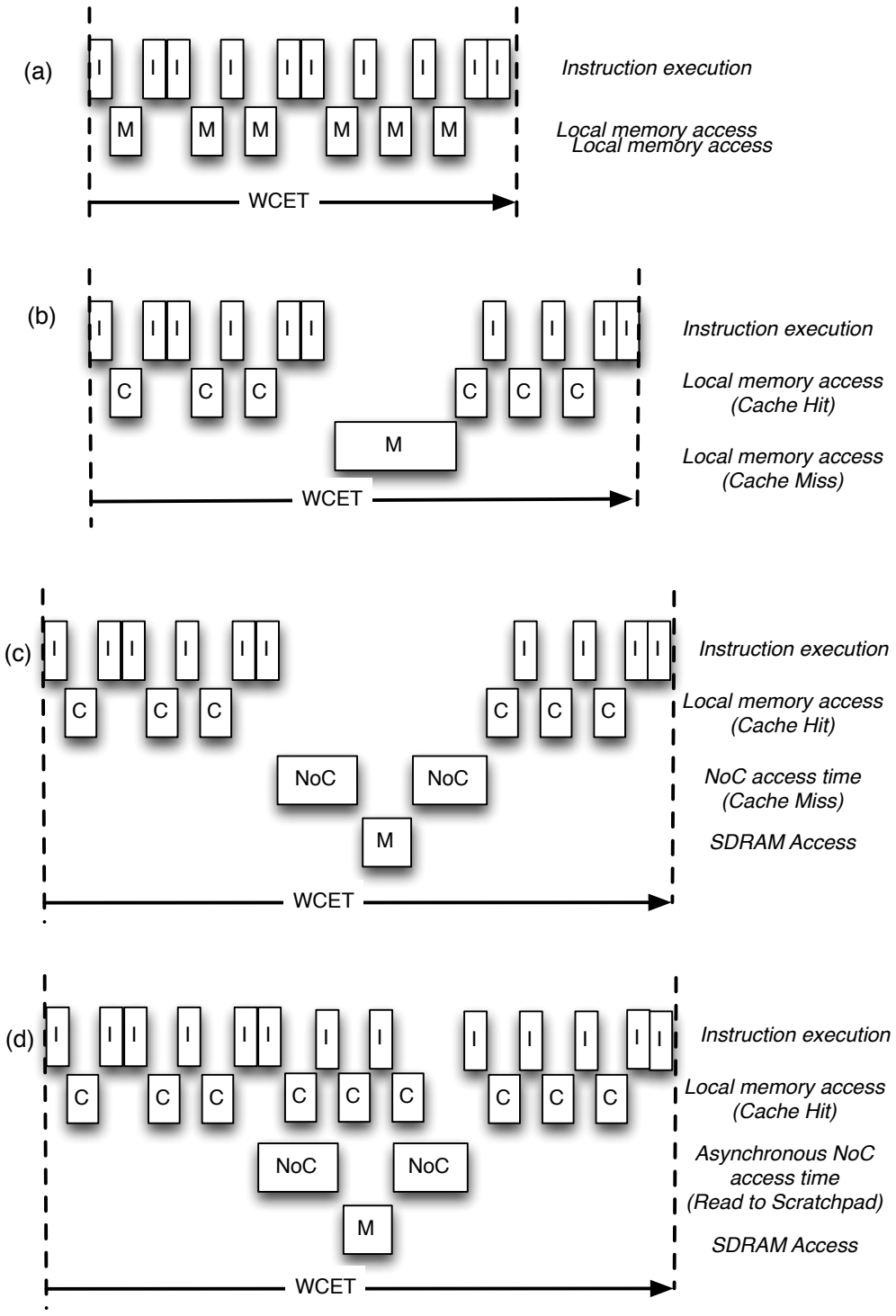

Figure 2: Memory Timing for Simple Uniprocessor (a), Shared Memory (b), NoC Memory (c)

bus connections to off-chip SDRAM, reconfigurability refers to two particular areas of the memory hierarchy:

1. Dynamic Memory Controller – to increase the flexibility of the main (off-chip) memory, T-CREST aims to develop a dynamically scheduled SDRAM controller (further details in section 3).

2. Scratchpad Memory – the inclusion of SPM within each NoC CPU tile allows software level control of a distributed SPM hierarchy whereby data can be moved between any pair of SPMs in the NoC (further details in section 2).

The two aspects of dynamic SDRAM controller for off-chip memory and flexible SPM within the NoC together constitute the T-CREST reconfigurable memory architecture.

2 On-chip Memory Hierarchy

This section deals with the on-chip *local* memory associated with each CPU. A number of alternative implementations are possible for this local memory. Regardless of implementation, the *purpose* of the local memory is always to minimise the amount of time in which the CPU is idle because it is waiting for information from some more distant memory, such as the off-chip *external* memory [11], described in Section 3.

2.1 Scratchpad Memory (SPM)

The simplest form of local memory is merely a small slice of physical memory, typically static RAM, placed close to the CPU. The address decoder and instruction/data bus are arranged so that accesses to a specific address range are routed to the local memory. This sort of local memory is termed a *scratchpad memory* (SPM) [15].

SPM is very simple and has minimal requirements in terms of silicon area and energy in comparison to other forms of local memory of identical size, since these always add some further mechanism [21].

However, those additional mechanisms may be useful, because SPM has the disadvantage that programs must explicitly control the resource. This is a software mechanism; the program, or compiler, must plan the usage of the physical memory. This would include (1) deciding which variables and functions should be placed in SPM during each part of the program, and (2) arranging to copy those variables or functions into place. This incurs three sorts of overhead:

1. Copying information to/from SPM incurs a time cost *in addition to* the time taken to transfer the data, because the copying operation is *not* a side-effect of some other instruction. Instead, the copy is explicitly directed by some sort of copying or pre-fetching instruction. There is an inevitable additional cost for executing that copying instruction.

2. If the SPM allocation is made *offline*, i.e. at compile time, it will match an *expected* or *typical* sequence of accesses to memory, rather than the actual sequence that will be generated at run-time. While such a *profile* can be an accurate representation of run-time operation, any inaccuracy caused by data dependence at run-time can lessen the efficiency of SPM. The allocation may be optimal with respect to the profile without being optimal with respect to all possible execution paths.

3. If tasks may be preempted, the SPM space will either need to be divided between the tasks offline (*static partitioning*) or divided at run-time. The former strategy limits the amount of space available to each task, while the latter incurs additional costs on each context switch due to the overhead of copying information to/from SPM.

2.2 Caches

The disadvantages listed above may be avoided by introducing *caching*, which is essentially a mechanism to adapt the contents of local memory to match the recent requirements of the program. The

basic physical SPM is extended with an additional *tag* memory, containing an index that relates each block of local memory to some block of external memory. This tag memory is an overhead: x bytes of *usable* local memory require approximately $\frac{5}{4}x$ bytes of storage when the tag memory is included[1].

Caches may be implemented using hardware or software. A *software* cache consists of a small function which is able to examine the tag memory to determine if a specified block of external memory has a copy in SPM. If so, then accesses to that block are routed to SPM: these are *cache hits*. If not, an older block of SPM is replaced with the new block: a *cache miss*. The process of servicing a hit is slow, because several instructions are required to carry out the comparison. Misses are even slower, incurring all of the costs of the hit in addition to the transfer time and the overhead of copying instructions. However, the software cache is able to dynamically adapt to execution paths. Software caches are typically used in systems where only an SPM is available, but it helps the programmer to introduce a cache for some data elements. A well-known example is the Cell CPU used in the Playstation 3 [14].

A *hardware* cache operates in parallel with the CPU, eliminating the overheads of the software cache, *except for* the physical time taken to transfer the data, and the space requirements for the tag store. No copying instructions are required. Instead, copying data from external memory is an implicit process triggered by any instruction causing a cache miss. The advantage of hardware caching versus software is speed; the disadvantage is an increase in the hardware required, which can be very small for a sufficiently simple design [12].

2.3 Time-predictability of Caches

For the purposes of creating a time-predictable architecture offering end-to-end guarantees, caches have the disadvantage of being difficult to predict, because their behaviour depends upon previous execution (*execution history*). *Worst-case execution time* (WCET) analysis must consider the worst-case scenario for cache contents at each point in the program. This incurs the following overheads:

1. The WCET estimate produced by analysis may be an overestimate due to imprecision in the knowledge of the cache state at each point in the program. WCET analysis tools must be pessimistic if information is not known. This is problematic for complex CPU architectures and cache architectures [18], but also for complex programs where data addresses are not predictable during offline analysis.

2. The worst-case scenario identified by WCET analysis may not be a result of the program itself, but rather some feature of the cache. For instance, two or more parts of the program (or variables) may *conflict* in cache by being allocated to the same block of physical memory [22]. The resulting WCET increase is not pessimism, but rather an accurate representation of the effect of the conflict.

SPM can avoid these overheads when used correctly, because conflicts may be resolved offline, and the act of mapping data to SPM provides a way to avoid the effects of unpredictable access sequences [22].

[1] Assuming 16-byte blocks and a 4-byte tag for each block: $\frac{16+4}{16} = \frac{20}{16} = \frac{5}{4}$.

2.4 Split Caches

To avoid some WCET overheads and the space overhead of the tag store, *split caches* are proposed [20]. In the proposed architecture, implemented as part of JOP [19], physically separate caches are used for different types of data, such as instructions (*methods*), stack data, heap data and constant data. These caches can be specialised to the sort of data stored within.

An important result of this specialisation is the simplification of the tag store. With the exception of the heap data cache, which is a small, fully-associative cache of conventional design, the new types of cache operate on a stack-frame basis, tracking only the amount of physical memory used by each function.

WCET analysis is also affected by this simplification, because cache misses involving instructions, stack data, and constant data can *only* occur when functions are called, or when functions return. Within a function, execution is as predictable as it would be if SPM were used.

However, split caches have certain disadvantages:

1. Methods can be large, and using a method (or its constant data, or its stack data) as the atomic unit of allocation brings an overhead through the possibility that code or data may be loaded, and will take up space, but will never be used. Thus, the physical space and transfer time are potentially under-utilised.

2. A *first-in first-out* (FIFO) buffer is used to maintain the list of elements stored in each cache. WCET analysis of caches managed by FIFOs is known to be difficult because the worst-case state of the FIFO is not easy to determine [17, 24]. WCET analysis for split caches deals with this by being pessimistic whenever there is a possibility that all methods will not fit in the FIFO [23].

3. The disadvantages of any other sort of cache are retained for heap data. Though a fully-associative cache design prevents conflicts, it also forces the cache to be small, making cache misses more likely. In order to permit WCET analysis, the cache must also adopt a write-through policy which slows down all stores [8]. It is for heap data that SPM will offer the greatest benefits.

2.5 SPM and Cache with Network-on-Chip

Local memory is local to each CPU core: a multicore system containing n CPUs will also contain at least n local memories. These will be linked to external memory via a shared bus or *network-on-chip* (NoC).

A simple arrangement would use the NoC purely for data transfers between external memory and local memory. This would create two problems:

1. A *bottleneck* due to the need to share limited memory bandwidth between n CPU cores.

2. A *coherence* problem between the local memories. Cache coherence issues occur when caches may be updated independently of external memory [11]. Each local memory may contain *stale* data that has been updated in another memory (external, or local to another CPU), but not yet evicted from this one. Large and complex mechanisms are needed to detect this situation.

It is unlikely to be feasible to use any sort of *cache coherence protocol* [11] on the T-CREST NoC. Such protocols do not scale very well and cannot support extremely large numbers of cores. It is also difficult to make useful guarantees about end-to-end timing when such a protocol is in effect; one is forced either to ensure that no *coherence conflicts* can occur (in which case, the protocol is unused) or to consider such conflicts throughout WCET analysis (in which case, the result is pessimistic).

Split caches ameliorate some of the problem. Caches for instructions and for constant data will not require any coherence protocol because the data within is read-only. Caches for stack data also do not require coherence protocols because the data is not shared with any other stack cache. However, the coherence problem remains for heap data, which is both read-write and may be shared between CPUs.

The easy solution is a simple change in the way the NoC is used, to allow data transfers *between* local memories *as well as* transfers between external memory and local memory. That way, external memory (and/or cache coherence protocols) do not need to be used for sharing data between CPU cores. Instead, that data can be efficiently transferred between the CPU cores across the NoC.

This mode of operation is not suited to any sort of cache because the transferred data does not have an address in external memory. However, it is suited to SPM. An existing mechanism for transferring data between external memory and SPM may be adapted to transfer data between one SPM and another - using the NoC as interconnect.

An alternative implementation could allow each CPU to send and receive NoC packets (*flits*) directly, but this would be a CPU-bound implementation which would reduce the amount of CPU time available to do useful work. SPM to SPM transfers is a natural way to offload the transfer onto memory-copying hardware while providing send and receive buffers in local memory.

2.6 SPMs within T-CREST

As previously noted, SPM may be used in place of any sort of cache with a resulting simplification of hardware. But this simplification comes at the cost of an additional transfer time overhead, and potentially less effective use of physical space. Though the operation of SPM is highly predictable, the operation of caches is also amenable to WCET analysis if the caches are appropriately designed.

The place where SPM really stands out as significantly better than cache are the cases of heap data and shared data, where it is a real benefit to (1) avoid a cache coherence mechanism, and (2) avoid the unpredictability of data-dependent conflict misses. However, within the T-CREST project, we consider it even more important to remain flexible, and therefore we propose the inclusion of both SPM and heap data caches as described in earlier work [20]. A method cache and stack cache are also used, but the main purposes of the SPM are (1) to act as an efficient alternative to the heap data cache, and (2) to enable direct data transfers between CPU cores. This would be a data SPM only, not storing any program instructions.

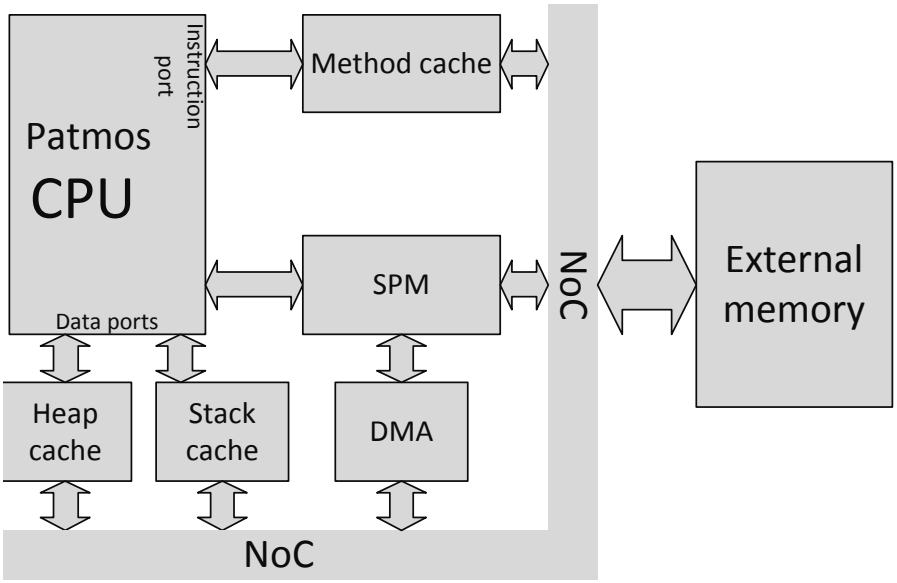

Figure 3: Proposed memory hierarchy for T-CREST.

A second use for SPM within T-CREST is for operating system (OS) software and supporting code, local to each CPU. The proposed Patmos architecture contains a number of complex functions, including those for managing the split cache, handling interrupts, and scheduling tasks. It may be useful to introduce an instruction SPM to hold the code for this functionality, particularly if this code must be able to operate without sending any request to external RAM.

2.7 DMA Controller

Aside from the hardware needed to implement the split cache and SPM, a *direct memory access* (DMA) controller is needed to transfer data to and from SPM (Figure 3). The SPM is linked to the NoC via the DMA controller, and directly to the CPU core's data port. A single physical memory port is used to implement these connections, with priority arbitration giving the CPU the highest priority.

The SPM is used only for storing heap data. DMA controller instructions are introduced to the CPU's instruction set in order to facilitate transfers to and from the SPM. These instructions are provided in two forms: blocking, and non-blocking.

Blocking DMA operations will be useful for code that needs to load new data into SPM before it can proceed. Non-blocking operations will be useful if the program is able to carry out some other operations before the data is needed. If data is being processed iteratively by a loop that passes over a much larger data structure than the available SPM space, an interleaving of computation and DMA transfers allows the next block of data to be fetched while the current one is being processed.

The operations could be generated automatically by a compiler, but it will be simplest to expose them to the programmer through the use of C macros and inline assembly code.

The SPM is addressable externally, with an address range that is unique on the NoC. This allows DMA operations initiated elsewhere on the NoC to use the SPM as a source or destination.

3 Off-chip Memory Hierarchy

This section discusses reconfiguration of the off-chip SDRAM memory. First in Section 3.1, we discuss the concept of use-cases and identify three important reconfiguration operations for the off-chip memory and explain their associated benefits. Three reconfiguration service classes for real-time systems considered in this project are introduced. Then in Section 3.2, we present the general architecture of a proposed reconfigurable real-time memory controller and a description of three particular architecture instances used in this project. Lastly, Section 3.3 explains important challenges with reconfiguration of the memory controller and presents the considered reconfiguration options for the three architecture instances.

3.1 Use-cases and Transitions

Complex systems often execute multiple applications at the same time and a set of such concurrently running applications are referred to as a *use-case*. The number of use-cases in different systems varies greatly, but is growing rapidly and is already in the hundreds for high-end televisions. This impressive growth is intuitively understood by considering that the number of possible use-cases in a system affording high application-level parallelism *increases exponentially* with the number of applications. In such systems, applications can be dynamically started and stopped at any time, triggering *use-case transitions*. This is shown in Figure 4 for a system supporting two applications running in parallel, where five use-cases labeled u_1 through u_5 are created as three applications start and stop their executions. Although some applications start and stop during a use-case transition, others may be continue executing and should do so in an oblivious manner. These applications are referred to as *persistent applications* [10] and are considered one of the main challenges of this project. As an example, the MP3 playback application in Figure 4 is persistent during the indicated use-case transition between u_3 and u_4, although all applications in the figure are persistent during at least one transition.

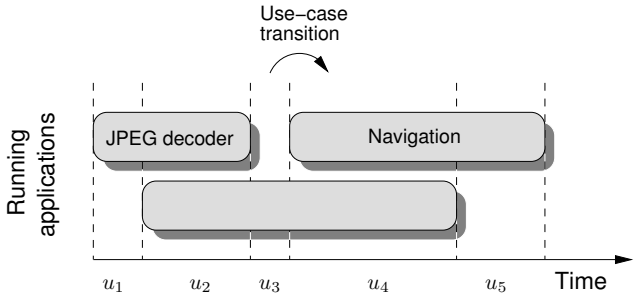

Figure 4: Starting and stopping applications triggers use-case transitions.

This project considers three primary *reconfiguration operations* for the memory controller. The first operation involves starting and stopping memory clients, while persistent memory clients continue accessing the memory. However, reconfiguration is also required for other reasons. Different use-cases may have diverse requirements in terms of bandwidth, latency, and power consumption. Efficiently satisfying these constraints requires the memory controller to be reconfigured to enable dif-

ferent trade-offs between these properties. Two mechanisms can be used by the memory controller to enable this trade-off, corresponding to the second and third reconfiguration operations. The second reconfiguration operation is *frequency scaling* the memory device, where lower frequency reduces power consumption and the guaranteed bandwidth and increases latency. The third considered reconfiguration operation is changing the *access granularity* of the memory controller (the size of the data blocks that are read/written to the memory per access), where larger granularity amortizes overhead cycles over larger data transfers and increases the guaranteed bandwidth and reduces latency [9] and energy consumption (although power consumption increases). Neither of these mechanisms has previously been reconfigured in real-time memory controllers with persistent memory clients.

The first challenge of reconfiguration is *functional correctness*, which means e.g. that no requests are lost, arrive at the wrong memory client, and that the memory controller does not deadlock. The context of time-predictable and composable real-time systems add additional challenges, since both the reconfiguration time of the system and the response times of persistent memory clients must be bounded and independent of other memory clients. In the T-CREST project, three different *reconfiguration service classes* are distinguished. They are listed here in order of ascending difficulty:

1. *Reconfiguration without persistent service* just requires the ability to start and stop memory clients in a functionally correct manner.
2. *Predictable reconfiguration with persistent service* requires that the response times of persistent memory clients are *bounded* before, during, and after reconfiguration in addition to functional correctness.
3. *Composable reconfiguration with persistent service* requires that the response times of persistent memory clients are not changed by a single clock cycle during or after reconfiguration in addition to functional correctness.

3.2 Reconfigurable Real-time Memory Controller

This section introduces the proposed architecture of a reconfigurable predictable and composable memory controller. It is based on the architecture in [2], but has been extended with additional configuration infrastructure to enable the individual blocks to provide different trade-offs between bandwidth, latency, and power consumption for each use-case. The proposed architecture, shown in Figure 5, comprises a resource *front-end* and an SDRAM *back-end*. The front-end is independent of memory technology and contains components to implement predictable and composable resource sharing [3]. The back-end interfaces with the actual memory device and controls it in a predictable and/or composable manner, depending on its configuration. We proceed by briefly explaining the blocks the front-end and back-end, respectively.

3.2.1 Resource Front-end

The front-end comprises three main simple and reusable blocks: *Atomizers*, *Delay Blocks*, and a *Resource Bus*. Additionally, there is a *Configuration Bus* that allows registers inside the different blocks

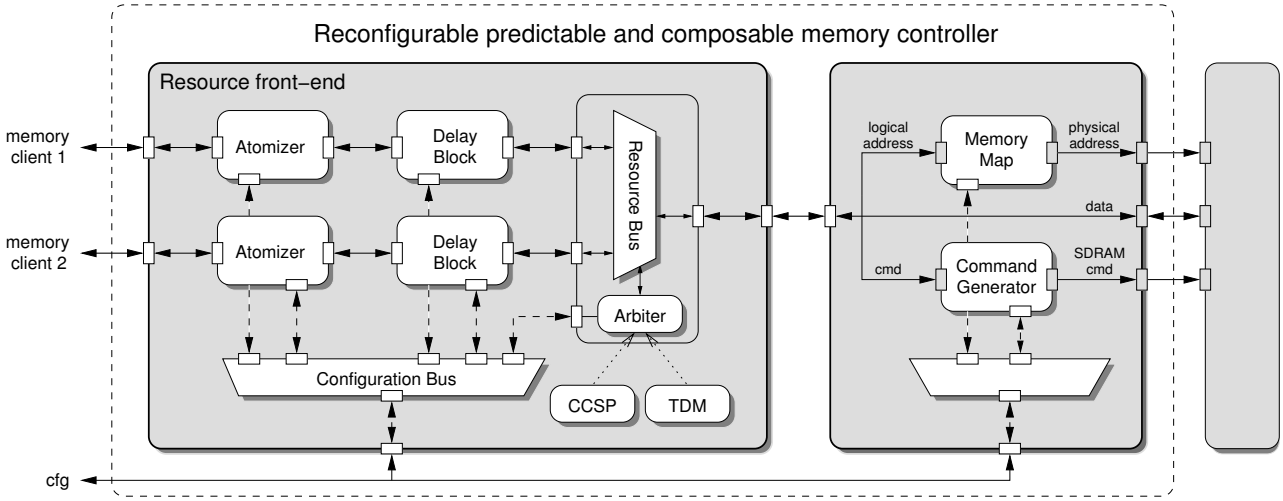

Figure 5: Architecture of reconfigurable predictable and composable memory controller.

to be programmed via memory mapped I/O during use-case transitions. The blocks communicate using a *device transaction level* (DTL) protocol [16], which is a standardized communication protocol similar to AXI. All white ports shown in Figure 5 are DTL ports, while other ports are heterogeneous custom interfaces.

The Atomizer is responsible for making the memory controller preemptive at a known granularity, which is required to implement composability in a robust manner [3]. This is achieved by splitting requests into *atomic service units*, referred to as atoms, which are served by the memory in a known bounded time. Large requests are hence chopped up in smaller pieces and their original sizes are stored in the Atomizer to allow it to merge responses back into the size expected by the memory client. The size of an atom is configured to match the access granularity of the memory, which is determined by the Memory Map in the SDRAM back-end. Reconfiguring the access granularity of the memory to provide a different trade-off between bandwidth, latency, and power, hence requiring both blocks to be reconfigured.

The purpose of the Delay Block is to make a predictable shared resource behave in a composable manner. This is done by hiding variation in temporal interference in the shared hardware blocks, which is achieved by delaying responses and flow-control signals to *emulate worst-case interference* from other memory clients. This provides a composable interface towards the Atomizer, making the interface of the entire front-end composable, since the Atomizer is not shared between memory clients [3]. The Delay Block consists of buffers to store requests and responses and logic that determines when to release responses and flow-control signals to emulate worst-case interference. These release times are use-case dependent and computed at run-time based on parameters that are programmed during use-case transitions.

The Resource Bus is a regular DTL bus that schedules requests, according to the policy of an attached predictable and/or composable arbiter. Examples of supported arbitration policies are Time-Division Multiplexing (TDM) or Credit-Controlled Static-Priority (CCSP) [6]. The configuration of the arbiter is use-case dependent and needs to be reprogrammed on use-case transitions.

3.2.2 SDRAM Back-end

The SDRAM back-end comprises two main functional blocks, the *Memory Map* and the *Command Generator*. The Memory Map translates the logical address of the request into a physical memory address, indicating the target bank, row, and column of the SDRAM device [13]. This translation can be done in a variety of ways, which impacts the exploited level of bank-parallelism in the memory [4]. Exploiting more bank-parallelism increases the guaranteed bandwidth by enabling overhead cycles in the memory to be hidden. However, this comes at cost of increased power consumption, since more memory banks have to be opened and closed per request [7]. The Memory Map is programmable to allow trade-offs between worst-case and average-case bandwidth, latency, and power to be made on use-case transitions. How the Memory Map is programmed determines the minimum amount of data that can be served efficiently, which determines the access granularity of the memory controller.

The final block of the back-end is the Command Generator, which generates a sequence of SDRAM commands that respects the timing constraints of the memory device. The sequence of commands depend on if the scheduled request is a read or a write and on the chosen memory map configuration. The current implementation of the Command Generator dynamically schedules *memory patterns*, which are pre-computed sequences of SDRAM commands. The memory patterns exist in six basic flavors: 1) read pattern, 2) write pattern, 3) read/write switching pattern, 4) write/read switching pattern, 5) refresh pattern, and 6) power-down pattern. An example of how requests maps to memory patterns is shown in Figure 6. The patterns are *automatically generated* [1] by a tool at design time based on the timing constraints of the particular SDRAM device and the chosen access granularity of memory. In this project, suitable patterns must be computed for all use-cases and programmed during use-case transitions if necessary. As a part of the T-CREST project, we will also look into increasing the dynamism in the predictable and composable by using a dynamic SDRAM command scheduler, instead of the currently employed pattern-based implementation. In this case, reconfiguration of the Command Generator is expected to reduce to only programming a few register values with information about the current memory map.

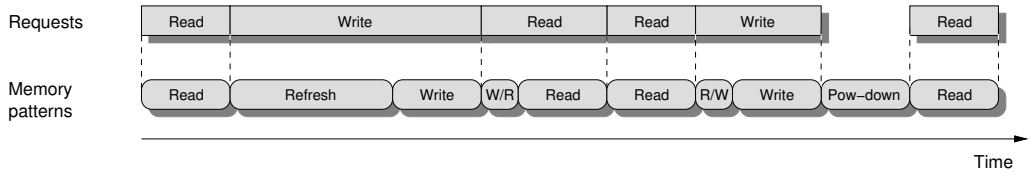

Figure 6: Mapping from requests to memory patterns.

3.2.3 Architecture Instances in T-CREST

In the T-CREST project, three instances of the predictable and composable memory controller are considered. The three instances differ in two important ways: 1) Whether or not the execution times of all requests are equal, which determines if the memory is a composable resource or just predictable [5]. The current implementation of the memory controller is just predictable, although it can be turned into a composable resource by adapting the tooling to compute composable memory patterns where reads and writes have the same execution time to reduce reconfiguration complexity.

However, this reduces the guaranteed bandwidth and increases power consumption and latency. This modification will be implemented as a part of this project. 2) Whether or not it uses TDM arbitration, which is an inherently composable arbitration policy. This makes it easy to add and remove memory clients while persistent memory clients execute, since reservations of different clients are independent. This makes resource reservation an application-level concern instead of a use-case concern, resulting in linear reconfiguration complexity of the arbiter instead of exponential. This has previously been exploited to enable independent and scalable reconfiguration of networks-on-chips [10]. TDM arbitration furthermore has the advantage that it does not require a Delay Block to implement composable resource sharing in case the memory itself is composable [5]. This further reduces the reconfiguration complexity, since there is one less block to safely reconfigure. The three considered architecture instances are listed here in ascending order of reconfiguration complexity:

1. *TDM arbitration with composable memory.* In this case, the execution times of all requests are equal, making the memory a composable resource. Sharing it with a TDM arbiter implies that the memory controller is composable without using the Delay Block.

2. *TDM arbitration with predictable memory and Delay Block.* In this case, the execution times of requests are not equal and a Delay Block is required to emulate worst-case interference in the memory, increasing configuration complexity.

3. *Any predictable arbiter with predictable memory and Delay Block.* The most general and complex architecture considered for the memory controller. This work will particularly consider the CCSP arbiter [6], since it is an example of a complex arbiter where the timing behavior of a memory client depends on the reservations of others.

3.3 Considered Reconfiguration Options in T-CREST

This section discusses the challenges associated with the three primary reconfiguration operations and discusses the theoretically supported reconfiguration service classes for the three considered architecture instances, previously presented in Section 3.1.

The challenge with starting and stopping memory clients without considering persistent clients is doing it in a functionally correct manner. The easiest way to guarantee that reconfiguration is done in a safe manner is to assert that the memory controller is in a *quiescent state*, which means that the memory controller is idle and does not contain any transactions. This could be implemented by letting the memory controller keep track of incoming and outgoing requests and provide state information to a reconfiguration task via the configuration ports on the front-end and back-end, respectively. Once the memory controller is quiescent, the settings of the arbiter and the Delay Block can be reprogrammed to reflect the new use-case.

Starting and stopping memory clients in a predictable manner with persistent clients adds the complexity of varying response times per use-case. However, this can be addressed by using the longest response time for any use-case in which a memory client is active for analysis, although more refined solutions may be possible and will be investigated. Composable reconfiguration with persistent memory clients is possible with TDM and a composable memory, as long as the reservations for the persistent clients do not change. More flexible and efficient schemes may be possible and will also

Reconfiguration class	No persistence			Persistent predictable			Persistent composable		
Operation	Start	Freq.	AG	Start	Freq.	AG	Start	Freq.	AG
TDM comp. patterns	yes	yes	yes	yes	yes	yes	yes	no	no
TDM & Delay Block	yes	yes	yes	yes	yes	yes	yes	yes	yes
CCSP & Delay Block	yes	yes	yes	yes	yes	yes	yes	yes	yes

Table 1: Theoretically supported reconfiguration options for the different architecture instances.

be investigated in this project. Composable reconfiguration is also possible for instances with predictable arbiters and Delay Blocks, as long as the Delay Blocks are programmed with the parameters corresponding to the use-case with the longest response time.

Frequency scaling the memory is primarily done by reprogramming the memory patterns used by the Command Generator in the back-end and can be done when the memory controller is in a quiescent state and poses no additional problems over starting and stopping memory clients as long as persistence is not considered. Frequency scaling the memory can be done in bounded time and supports predictable reconfiguration with persistent memory clients. However, frequency scaling the memory changes the guaranteed bandwidth of the memory controller, potentially also requiring the bandwidth reservation in the arbiter to be reconfigured. If the combination of frequency scaling and bandwidth reservations results in different response times for different use-cases, the longest response time must be used for analysis. Composable frequency scaling is not possible for the architecture instance without a Delay Block. This is because frequency scaling implies changing the length of the memory patterns, which makes the memory a non-composable resource by definition. This is not a problem for instances with a Delay Block, provided that it is programmed with the parameters of the use-case with the longest response time.

Reconfiguring the access granularity of the memory controller is similar to frequency scaling. It primarily involves reprogramming the Memory Map and the Atomizer, but just like frequency scaling it affects the guaranteed bandwidth and may require reconfiguration of the arbiter. Just like frequency scaling, changing the access granularity affects the response times of the memory clients, but the same solutions apply to support predictable reconfiguration with persistent memory clients. An added challenge with predictable reconfiguration of the Atomizer when considering persistent clients is that it must be reconfigured between atoms. This means that the bound on reconfiguration time depends on both the production and consumption of data by the memory client, making it more challenging to derive. Deriving this bound is a part of this project. Similarly to frequency scaling, changing the access granularity of the memory cannot be done in a composable manner without a Delay Block, since this changes the lengths of patterns and makes the memory a non-composable resource. Composable reconfiguration of the Atomizer with persistent clients comes with an additional challenge also for instances with a Delay Block, since the data of a larger atom is not released identically to the data of many smaller atoms. This problem will also be addressed by this project.

Table 1 concludes this section by summarizing which reconfiguration options are theoretically supported for the different architecture instances and reconfiguration service classes, along with some corresponding assumptions. The table covers three reconfiguration operations for every combination of reconfiguration and architecture instance: 1) starting and stopping memory clients (start), 2) frequency scaling the memory device (freq.), and 3) changing the access granularity of the mem-

ory controller (AG). Note that some of the theoretically supported options may not be required or practical to implement, potentially omitting them from the final set of implemented options.

4 Discussion

This document has discussed the proposed conceptual aspects of the *memory hierarchy* within T-CREST. Initially, bounded end-to-end timing of software processes with different memory hierarchies were discussed in order to set a context for the overall concepts of the T-CREST memory hierarchy. Then, the concepts of the T-CREST reconfigurable memory hierarchy were discussed:

1. Dynamic Memory Controller (off-chip memory) – to increase the flexibility of the main (off-chip) memory, T-CREST aims to develop a dynamically scheduled SDRAM controller.
2. Scratchpad Memory (on-chip memory) – the inclusion of SPM within each NoC CPU tile allows software level control of a distributed SPM hierarchy whereby data can be moved between any pair of SPMs in the NoC.

The two aspects of dynamic SDRAM controller for off-chip memory and flexible SPM within the NoC together constitute the T-CREST reconfigurable memory architecture.

In addition to further refinement of the memory hierarchy concepts in line with other work packages (i.e. processor, NoC, compiler, worst-case execution time), Work package 4 (Memory Hierarchy) now focuses on three tasks:

- Reconfigurable Real-Time Memory Architecture Implementation (Task 4.2)
- Dynamic memory controller development (Task 4.3)
- Feedback Control-Based Memory Scheduling (Task 4.4)

The final task considers more flexible use of the entire memory hierarchy by applications that require predictable overall timing. By actively tuning memory scheduling at run-time via an adaptive (feedback-inspired) mechanism, dynamic improvements to the performance of the memory hierarchy will be made whilst maintaining timing guarantees.

References

[1] B. Akesson, W. Hayes, and K. Goossens. Automatic generation of efficient predictable memory patterns. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 1, pages 177 –184, August 2011.

[2] Benny Akesson and Kees Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2011.

[3] Benny Akesson, Andreas Hansson, and Kees Goossens. Composable resource sharing based on latency-rate servers. *Euromicro Symposium on Digital Systems Design*, pages 547–555, 2009.

[4] Benny Akesson, Po-Chun Huang, Fabien Clermidy, Denis Dutoit, Kees Goossens, Yuan-Hao Chang, Tei-Wei Kuo, Pascal Vivet, and Drew Wingard. Memory Controllers for High-Performance and Real-Time MPSoCs. In *CODES+ISSS: Proceedings of the IEEE/ACM international conference on Hardware/software codesign and system synthesis*, October 2011.

[5] Benny Akesson, Anca Molnos, Andreas Hansson, Jude Ambrose Angelo, and Kees Goossens. Composability and predictability for independent application development, verification, and execution. In Michael Hübner and Jürgen Becker, editors, *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, chapter 2. Springer, December 2010.

[6] Benny Akesson, Liesbeth Steffens, Eelke Strooisma, and Kees Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2008.

[7] K. Chandrasekar, B. Akesson, and K. Goossens. Improved power modeling of ddr sdrams. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 99 –108, 31 2011-sept. 2 2011.

[8] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. In *Proceedings of Embedded Real Time Software and Systems*, May 2010.

[9] Sven Goossens, Benny Akesson, and Kees Goossens. Memory-Map Selection for Firm Real-Time Memory Controllers. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2012.

[10] Andreas Hansson, Martijn Coenen, and Kees Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 954–959, 2007.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach: Third Edition*. Morgan Kaufmann Publishers Inc.

[12] Mark D. Hill. A case for direct-mapped caches. *Computer*, 21(12):25–40, 1988.

[13] B. Jacob, S.W. Ng, and D.T. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub, 2007.

[14] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. R&D*, 49(4/5):589–604, 2005.

[15] P. Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., 2006.

[16] Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, 2002.

[17] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[18] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, July 2006.

[19] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[20] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16, Tokyo, Japan, March 2009. IEEE Computer Society.

[21] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proc. DATE*, page 409, 2002.

[22] Jack Whitham and Neil Audsley. Investigating average versus worst-case timing behavior of data caches and data scratchpads. In *Proc. ECRTS*, ECRTS '10, pages 165–174, 2010.

[23] 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.

[24] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, July 2009.