T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 4.2 Reconfigurable Memory Controller Design and Implementation

Version 1.0
28 February 2013
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 João Baptista Av. D. Joao II, Torre Fernao Magalhaes, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: joao.baptista@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail: silvia.mazzini@intecs.it
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

1 Introduction **2**

2 On-Chip Memory Architecture Implementation **3**

2.1 T-CREST Architecture Overview . . . 3
2.2 Shared Memory Tree Framework . . . 4
2.3 Blueshell CPU Tile . . . 5
2.3.1 Local Memory . . . 5
2.3.2 Cache Design . . . 6
2.3.3 Avoiding Read Hazards . . . 7
2.3.4 Cache Control Details . . . 8
2.3.5 Scratchpad Memory (SPM) . . . 8
2.4 Tree Links . . . 10
2.5 Tree Routers . . . 10
2.6 Smarter Routers . . . 11
2.7 Other Sorts of Routers . . . 12
2.8 The Root of the Tree . . . 12
2.9 Status . . . 13

3 Off-Chip Memory Architecture Implementation **14**

3.1 Use-Cases and Transitions . . . 14
3.2 Background . . . 15
3.2.1 SDRAM Specifics . . . 15
3.2.2 Real-Time SDRAM Controller . . . 16
3.2.3 Latency-Rate Servers . . . 17
3.3 Architecture of the Reconfigurable Controller . . . 18
3.3.1 Resource Front-End . . . 18
3.3.2 SDRAM Back-End . . . 19
3.3.3 Architecture Instances in T-CREST . . . 20
3.4 Approach to Real-Time SDRAM access . . . 21
3.4.1 TDM Arbitration . . . 21
3.4.2 Composable Patterns . . . 23
3.5 Reconfigurable TDM Arbitration . . . 24
3.5.1 TDM Slot Allocation . . . 25

3.5.2 Predictable TDM Reconfiguration 25
3.6 Implementation 27
3.6.1 Design-Time Tool Flow 28
3.6.2 Run-Time Software 28
3.6.3 Run-Time Hardware 29
3.7 Experimental Results 30
3.7.1 Experimental Setup 30
3.7.2 Performance of Composable Patterns 30
3.7.3 Temporal Application Behavior 31

4 Requirements **36**

5 Conclusions **37**

Document Control

Version	Status	Date
0.1	First draft	31 January 2013
0.2	Second draft	19 February 2013
0.3	Third draft	22 February 2013
1.0	Final version	28 February 2013

Executive Summary

WP4 (Memory Hierarchy) has the main objective of specification, design and implementation of a reconfigurable memory hierarchy that provides time-predictable performance. This 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 4.2 (Reconfigurable Real-Time Memory Architecture Implementation), due 18 months after project start as stated in the Description of Work. It discusses the implementation of the *memory hierarchy* with T-CREST. The concepts and overall design of the memory hierarchy were delivered within D4.1 (Reconfigurable Memory Architecture Specification).

This document discusses the memory architecture implementation in two main areas:

1. On-Chip Memory Architecture – considering the inclusion of scratchpad memories within NoC CPU tiles, communication between scratchpad memories via the NoC architecture, and the communication between NoC CPU tiles and the off-chip memory controller.
2. Off-Chip Memory – considering off-chip (DDR) memory controllers.

1 Introduction

The T-CREST platform consists of CPU tiles linked by two separate mechanisms (Figure 1):

1. *Network-on-Chip (NoC)*: conventional *mesh* NoC providing communication between CPU tiles. The NoC is being developed in WP3 and the CPU tile within WP2.
2. *Shared Memory Tree*: providing access to a shared, off-chip memory from each CPU tile.

Hence the three major components of the T-CREST memory hierarchy are:

1. local memories within the CPU tile;
2. shared memory tree providing connectivity between CPU tiles and
3. the controller for off-chip memory.

In this document, we describe all implementation details of the above, noting that the basic discussion of the memory hierarchy was presented in D4.1 ("Reconfigurable Memory Controller Concepts").

Specifically, section 2 describes with the shared memory tree. Section 3 deals with the off-chip memory controller. This is linked to each CPU tile via the shared memory tree and provides access to a large shared memory space.

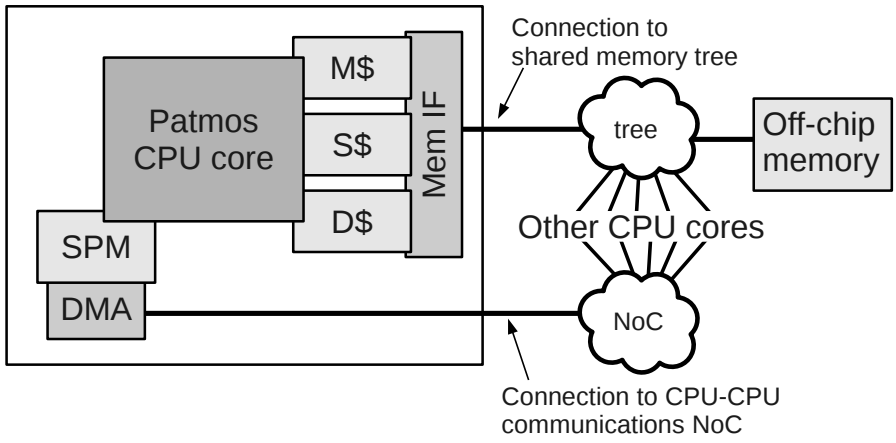


Figure 1: Overview of the T-CREST platform: the CPU tile, the NoC connecting CPUs, and the shared memory tree for access to shared off-chip memory.

2 On-Chip Memory Architecture Implementation

This section deals with the implementation of the on-chip memory architecture of the T-CREST platform, dealing particularly with the shared memory tree and the memory elements of the CPU tiles (*scratchpad memories* (SPMs) and caches).

2.1 T-CREST Architecture Overview

As illustrated by Figure 1, the T-CREST platform features two connectivity mechanisms: a tree for access to shared memory, and a conventional NoC for communication between CPUs [16]. The latter is a two-dimensional grid of CPU tiles. Each tile can send messages to its neighbors to the north, east, south and west via network links [18]. The resemblance to the roads and blocks of a city leads to the name "Manhattan grid" (Figure 2). The NoC is being developed within WP3 [16].

The T-CREST project is utilizing both a NoC and a shared memory tree to provide the predictability required for both shared memory access due to CPU cache miss accesses and inter-CPU data transfers between application tasks. Whilst many other NoC designs merely rely upon a conventional Manhattan grid NoC for all communication, their *modus operandi* (and assumed computational model) tends to preclude cache memory accesses to shared memory (i.e. all the code and data required by each application task is kept entirely within local memory inside the CPU tile). The T-CREST project challenges the limitations of this assumption and enables CPUs to utilize shared memory for cache access. If off-chip memory accesses take place via the NoC (by encoding load and store requests as network messages), certain network links will become bottlenecks, particularly those close to the off-chip memory controller – hence the T-CREST approach for a separate shared memory tree.

For access to shared memory, a tree-shaped network is preferable – shown in Figure 1. The shared memory is placed at the root, and CPUs are placed at the leaves. Non-leaf, non-root tree nodes act as routers and multiplexers. In a simple implementation, each link in the tree provides the same bandwidth (Figure 3(a)), but it is quite easy to transition to an improved implementation where each link provides bandwidth appropriate to the amount of data to be carried (Figure 3(b)). This is called a *fat tree* as the bandwidth of a link is proportional to its physical bus width, and some links are "fatter" than others.

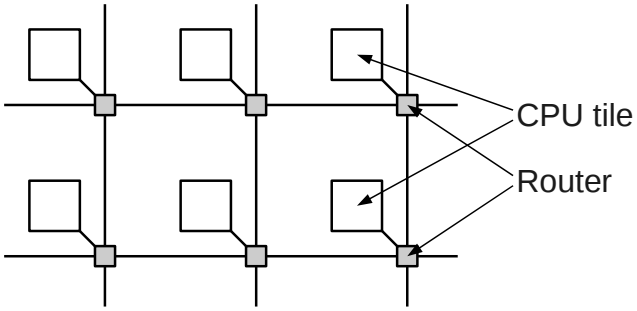


Figure 2: The T-CREST NoC – used for communication between CPUs [16].

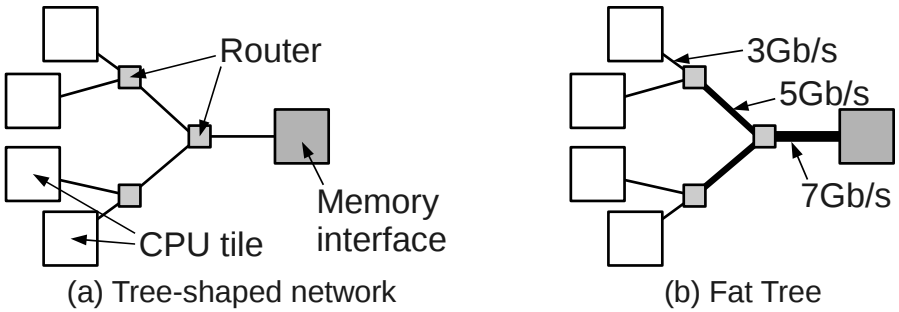


Figure 3: (a) A tree-shaped network-on-chip is preferred for access to shared memory. (b) Within a fat tree, links provide different bandwidth depending upon expected utilization.

2.2 Shared Memory Tree Framework

The T-CREST work plan includes Task 4.2: implementation of the memory architecture for the T-CREST platform. Further work on the integration of the on-chip memory with the NoC (WP3) and Patmos CPU (WP2) will be reported within deliverable D4.5 at month 30.

The implementation of the shared memory tree is termed XPortMC, and has been implemented and prototyped on FPGA – this is illustrated in Figure 4. In order to test XPortMC, the tree has been incorporated within the Blueshell NoC research framework developed (outside T-CREST) at the University of York. Key components include:

- *Blueshell CPU Tile* – off-the-shelf CPU core (Microblaze [21] developed by Xilinx) within an appropriate CPU tile;
- off-the-shelf external memory controller (developed by Xilinx);
- NoC mesh to connect CPU tiles (developed outside of T-CREST).

Blueshell's name is derived from the *Bluespec System Verilog* (BSV) language used for implementation [9].

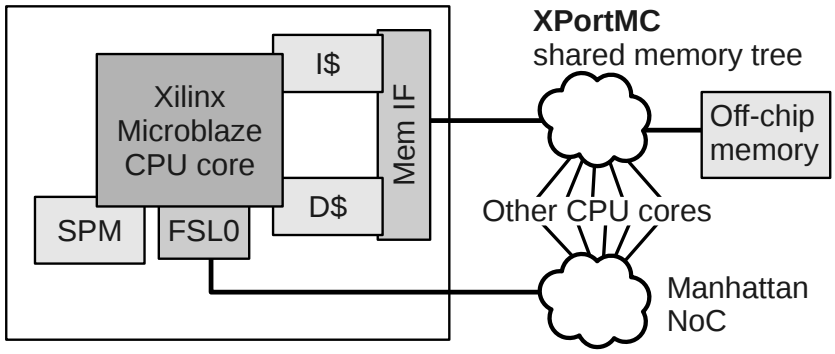


Figure 4: Blueshell includes prototype implementations of all three major components of the T-CREST platform: the shared memory tree, the Manhattan NoC, the CPU tile and the off-chip memory controller.

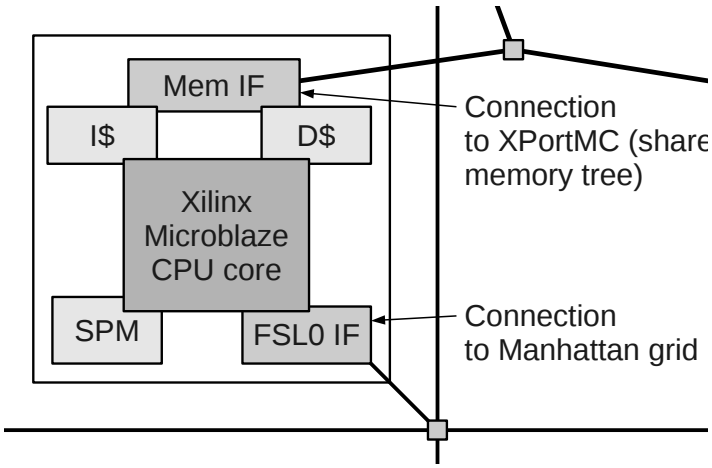


Figure 5: The Microblaze CPU tile has two external connections: one for off-chip memory access via the XPortMC shared memory tree, and another for the Manhattan NoC.

2.3 Blueshell CPU Tile

The leaf nodes of the shared memory tree are connected to the memory access ports of CPU tiles. Ultimately, the connection is for the Patmos CPU developed as part of T-CREST for WP2. Until this is fully complete, we made use of the Xilinx Microblaze CPU as the basis for the prototype CPU tile (Figure 5). The off-the-shelf Xilinx IP core is reused and wrapped in hardware designs developed for Blueshell. These provide the interface to both the Manhattan grid and the memory-access tree. The Blueshell CPU tile is architecturally equivalent to Patmos so that Patmos can replace it at a later date.

2.3.1 Local Memory

To be consistent with the overall T-CREST memory architecture as described in deliverable D4.1, the *internal* memory of the Blueshell Microblaze CPU tile is similar to Patmos.

The components of the internal memory architecture are *scratchpad memory* (SPM) and cache (Figure 5). The sizes are configurable at synthesis time. The defaults are an 8Kbyte scratchpad memory (SPM), an 8Kbyte instruction cache (I$) and an 8Kbyte data cache (D$). The SPM may be used for instructions and data and responds to all accesses in a single clock cycle.

The caches also respond within a clock cycle in the event of a cache hit. They are direct-mapped, write-through and do not allocate on write. They have a configurable block size that is set to 64 bytes by default.

Once development is complete, the Patmos CPU tile will include caches of unconventional types, notably a method cache (M$) and stack cache (S$). But these require support from programs, which must be specially compiled to use these features, and modifying the Microblaze C compiler to make use of M$ and S$ is beyond the scope of this T-CREST work package. Therefore, the Microblaze CPU tile uses only conventional cache designs, and runs programs produced with the off-the-shelf C compiler.

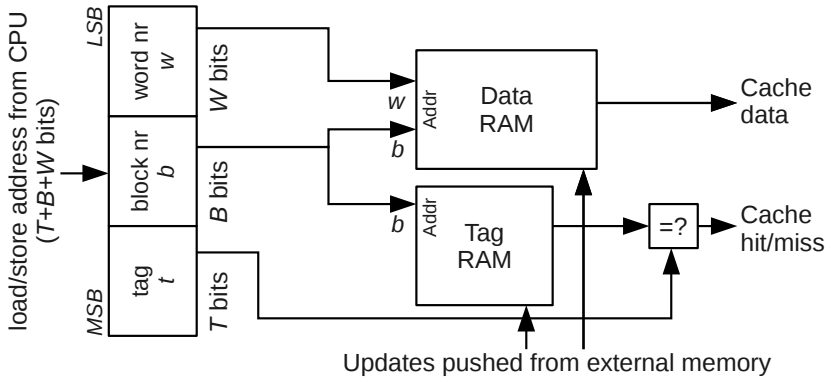


Figure 6: The internal layout of each cache. Caches are direct-mapped, write-through and no allocate on write: this is the same design as the built-in Microblaze caches [21].

2.3.2 Cache Design

A key feature within the T-CREST memory hierarchy concepts reported in deliverable D4.2 is the ability for the off-chip memory to push code / data to Patmos CPU tiles to provide pre-fetch to increase performance without decreasing predictability (this work is to be developed within task 4.4 of WP4). The initial implementation of this feature within the shared memory tree requires support within the CPU tile.

The cache push feature is supported by decoupling the mechanism for detecting/requesting cache misses from the mechanism for receiving data from off-chip memory. The mechanism has been tested by incorporating changes within the Blueshell CPU tile. We note that the required modifications cannot be made directly to the built-in Microblaze CPU caches, so it was necessary to reimplement those caches. Similar modifications might be made to the Patmos data cache at a later stage of the T-CREST project.

Figure 6 shows the internal layout of each cache. Addresses are generated by the CPU and then separated into a tag number t (T bits), a block number b (B bits) and a word number w (W bits). In the default configuration, the block size is $2^W = 64$ bytes, so $W = 6$. The cache size is $2^{B+W} = 8192$ bytes, so $B = 7$. As the CPU has a 32-bit address bus, the total $W + B + T = 32$, so $T = 19$.

The cache serves operations requested by the CPU and operations requested via the memory port. The CPU may request load and store operations (instruction fetches being equivalent to loads). The memory port may be used to update a cache block within the cache.

From the CPU, there are two operations (load and store) and two cases for each (hit and miss):

- *Load, hit.* Hits are detected by comparing the T uppermost bits of the address with the output of the tag RAM (Figure 6). If the two match, then the access is a hit. The data RAM output is forwarded to the CPU. As soon as a hit is detected, the CPU may issue another access. The total latency is a single clock cycle.
- *Store, hit.* Hits are detected as for loads. The data RAM is dual-ported, and the second port is used to update the RAM contents with the data word from the CPU. The latency is one clock cycle, but because the update takes place after hit detection, the cache cannot be immediately

reused for a load access. Loads are blocked for a short time by a hazard avoidance mechanism (described below). Stored data is always sent to off-chip memory.

- *Store, miss*. If a store operation is a miss, it is simply sent to off-chip memory, and no cache update takes place. This is due to the "no allocate on write" cache policy, also used by Microblaze's built-in caches [21].
- *Load, miss*. When a miss is detected, the cache stalls. All accesses are locked out while the miss is served. A request for the missing data is sent to off-chip memory, and the cache hardware repeatedly re-checks the tag RAM to see if the data is available yet. This is required for the push feature. There is no guarantee that the next cache block written to the memory port will be the cache block that is missing. It may be some other cache block, which was requested earlier or was pushed as a result of some prefetching process. In this case, the cache mechanism must continue to wait for the arrival of the correct block.

2.3.3 Avoiding Read Hazards

The RAM within the cache may be updated in two different ways: firstly as a result of stores issued by the CPU, and secondly as a result of data arriving from off-chip memory. In both of these cases, there is the possibility of a hazard, as a load operation may occur while the cache is being updated.

The cache implementation includes a hazard avoidance mechanism based around a counting semaphore. The counting semaphore acts as a mutual exclusion lock. The value is initially zero. It is incremented whenever an operation that updates the cache RAM begins. It is decremented whenever such an operation completes. Load operations are stalled unless the semaphore's value is zero.

This avoids the possibility that a load operation may be incorrectly treated as a hit due to the arrival of new data. If this did occur, the most likely result would be that the load operation would obtain invalid data. Therefore, no load operation takes place while the cache RAM is being updated.

Figure 7 shows the effect of this policy on the access times for the cache. It is not easy to trace low-level operations on the FPGA implementation of Blueshell, but the Xilinx ISim software can be used to simulate the FPGA implementation and capture the status of the CPU tile at a clock-cycle level.

As all of the accesses are hits, Figure 7 shows a latency of one clock cycle per access, e.g. nine clock cycles for nine load instructions (Figure 7(a)). The exceptions are:

- Where store operations must be blocked because earlier store operations have not yet been committed to off-chip memory. In Figure 7(b), the first store instruction does not block, but subsequent ones involve a single clock cycle of blocking time. This is because of the need to send both an address and written data to off-chip memory.
- Where a load operation must be blocked in order to avoid a hazard, because a store operation has not yet been committed to the data RAM. This is observed in Figure 8, where a store instruction is placed in the middle of eight load instructions. This store instruction completes after one clock cycle, but the following load instruction blocks for four cycles in order to ensure that the store is committed to cache and the contents are updated and accessible.

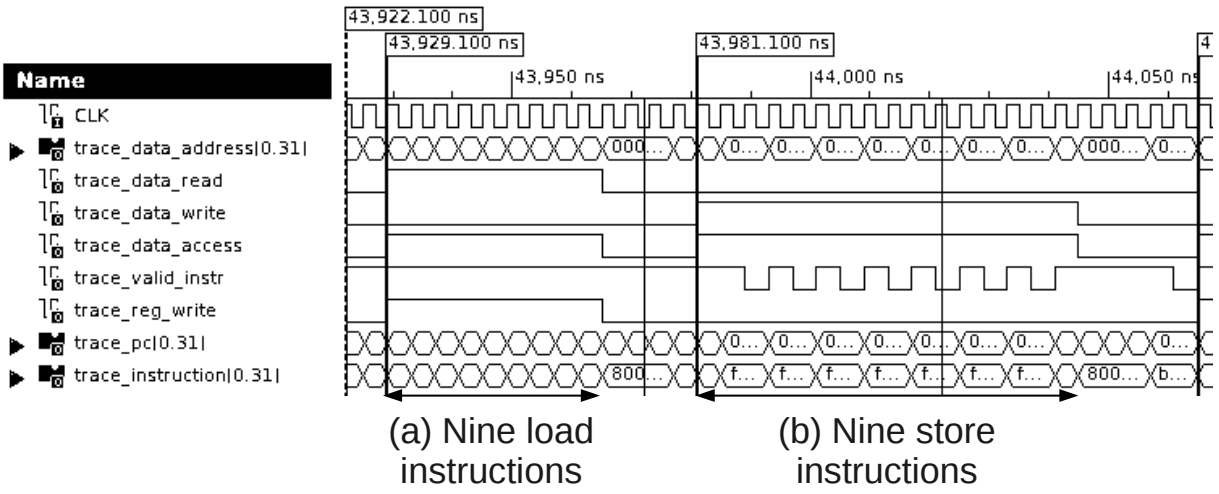


Figure 7: Xilinx ISim trace for a program consisting of (a) nine consecutive load instructions, all cache hits, and (b) nine consecutive store instructions.

These two situations are easily predictable by static analysis of the code as they are entirely determined by the operations immediately preceding each load or store operation. Moreover, they are typical of cache designs, as hazards must always be detected and avoided. The situations are not observed with the instruction cache as it never carries out any store operation.

2.3.4 Cache Control Details

It is useful to be able to invalidate cache lines from software. For example, this should be done whenever new code has been loaded into memory, e.g. by a bootloader that receives a program via a serial cable. The process of storing this new program does not update the instruction cache, so that cache may contain stale instructions from earlier execution. It must therefore be invalidated before starting the new program.

The caches in our test implementation within the Blueshell CPU tile allows cache blocks to be invalidated by sending addresses to Microblaze's second co-processor port, FSL1. Each invalidation command stalls the cache for a clock cycle.

2.3.5 Scratchpad Memory (SPM)

Ultimately, the T-CREST design aims to permit inter-CPU communication via a Manhattan grid network-on-chip. This communication could take two forms. Messages may either be sent directly by a CPU, or written to *scratchpad memory* (SPM) attached to a CPU and then sent via a *direct memory access* (DMA) controller.

At present, our implementation only supports the first mode of operation, i.e. network packets are sent directly by the CPU. The `put` instruction is used [21]. For example, the following code sends a network packet consisting of two words `HEADER0` and `HEADER1`[1]:

[1]This is Microblaze assembly code [21].

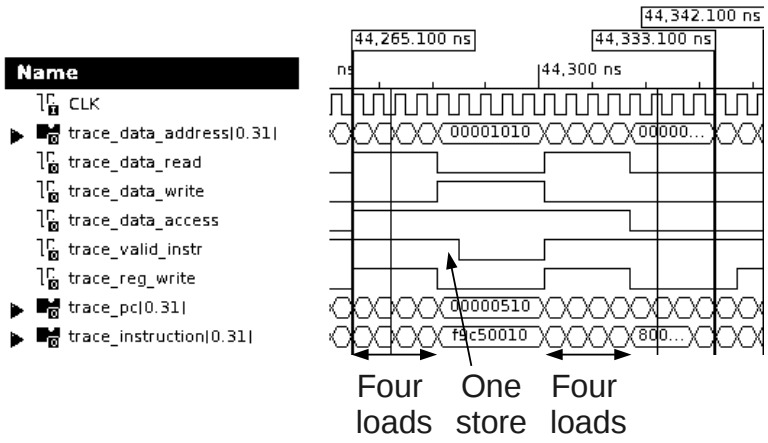


Figure 8: Xilinx ISim trace for a program consisting of four load instructions, one store instruction, and four more load instructions. The store instruction creates a hazard which means the subsequent load instruction is blocked for four clock cycles.

```
    addi    r5, r0, HEADER0
    addi    r6, r0, HEADER1
    put     r5, rfsl0
    put     r6, rfsl0
```

This network packet will be received by some other CPU using the `get` instruction.

Packets may be stored temporarily in SPM, and sent from SPM by a combination of load and `put` instructions. For instance, suppose that the CPU should send a packet of size `r5` words starting at address `r7`. The following assembly code may be used to send the packet:

```
0:  addi    r5, r5, -1
    lwi     r6, r7, 0       # load, r6 := [r7]
    put     r6, rfsl0       # send word
    bneid   r5, 0b
    addi    r7, r7, 4
```

Each SPM is currently only accessible from the attached CPU, which may use it to store code and data. Software running on the CPU must send and receive network packets and load/store them in SPM.

However, our CPU tile design makes it possible to extend the current implementation with a DMA controller to enable direct transfers between SPMs. The main difference between such a design and the current implementation is that at least one of the SPM's ports must be available for access by the DMA controller. This may mean that the SPM cannot be used to store instructions, or that a separate SPM is used for that purpose, or that one port is multiplexed between the DMA controller and the CPU.

2.4 Tree Links

The CPU tile accesses off-chip memory via a tree (Figure 3). The tree must contain at least two nodes: a root and a leaf. Each CPU is a leaf node and off-chip memory is the root node.

The link between these nodes is a full-duplex bus of configurable width (32 bits by default). In both directions, the bus structure is as follows[2]:

```
typedef struct {
    XPortMCControl  control;
    Bit#(32)        data;
    Bit#(4)         side;
    CPUIdBits       cpu_id;
} XPortMC deriving (Bits, Eq);
```

There are two types of packet:

- *Read* packets consist of a single `XPortMC` word with `control = C_READ`. These are generated by leaf nodes and sent towards the root. In these packets, `data` is the memory address where the read should begin, and `side` contains the number of words to be read, with `side` $= 0$ representing sixteen data words. The off-chip memory should respond to a *Read* packet by sending the requested number of words in a *Write* packet.
- *Write* packets consist of a header word with `control = C_WRITE` followed by 1-16 data words with `control = C_PAYLOAD`. In the header word, `data` is the memory address where the write should begin, and `side` contains the number of words to be written, with `side` $= 0$ representing sixteen data words. For each data word, `side` contains byte-lane enable signals. *Write* packets may be generated by either the root (when responding to a *Read*) or by leaves (when the CPU writes to memory).

Each `XPortMC` link is intended to be implemented by a FIFO buffer. This FIFO may cross clock domains so that links closer to the root provide more bandwidth. The disadvantage of using a FIFO is that crossing each link requires at least one clock cycle.

2.5 Tree Routers

The tree may be expanded by replacing any leaf with a subtree (Figure 9). The new component is a router, which grants each of the new leaves access to off-chip memory.

For packets sent from leaf to root, the router acts as an arbiter, routing complete packets from one child at a time. Packets cannot be fragmented, so the arbitration decision must be made as soon as a header word is received. Various arbitration policies are possible, such as statically prioritizing one child over another, or prioritizing packets depending on their source or content.

When a packet from a leaf is received, the `cpu_id` bit field may be modified to indicate the reverse path to subsequent routers and the root node. For instance, if the router has two child nodes, packets from child 0 may be modified as follows:

[2]The *hardware description language* (HDL) used is Bluespec System Verilog [9].

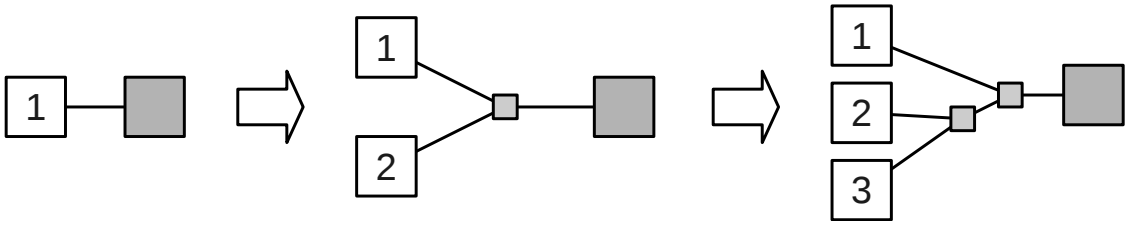


Figure 9: Expanding the memory access tree by adding routers. Each router allows a leaf node to be replaced by a subtree.

```
rule relay_up_0 if (arbiter == CHILD_0);
    XPortMC p = child0_in.first();
    child0_in.deq();
    p.cpu_id = p.cpu_id << 1;
    p.cpu_id[0] = 1'b0;
    parent_out.enq(p);
endrule
```

When a packet from the root is received, the opposite process is applied to determine where to send the packet:

```
rule relay_down;
    XPortMC p = parent_in.first();
    parent_in.deq();

    Bit#(1) mux = p.cpu_id[0];
    p.cpu_id = p.cpu_id >> 1;
    if (mux == 0) begin
        child0_out.enq(p);
    end else begin
        child1_out.enq(p);
    end
endrule
```

The default size of `cpu_id` is 10 bits, giving a maximum tree depth of 10 routers.

Each router does not need to run in the same clock domain as those around it, as the FIFO links between routers may cross clock domains. This means that some routers and some links may provide more bandwidth than others by operating at higher frequencies.

2.6 Smarter Routers

A basic router design might only include an arbiter, enforcing some arbitration policy to prioritize packets from its children.

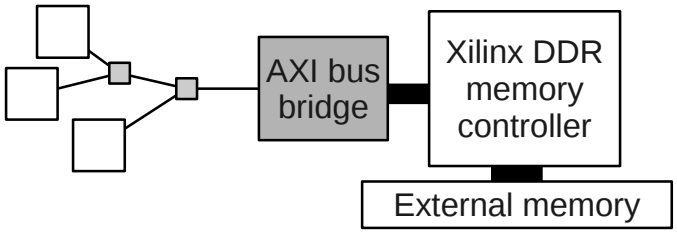


Figure 10: The node at the root of the memory access tree is implemented as a bridge to an AXI bus, enabling the use of off-the-shelf memory controllers.

Smarter designs might add further capabilities, and one T-CREST plan involves introducing routers that prefetch data and push it towards CPUs. These routers may have only a single child, but they will be capable of generating *Read* packets in response to other *Read*s.

For instance, a program may wish to iterate through an array of integers starting at address 1000 and ending at address 9000. This program uses the Manhattan grid to send a message to the prefetching router, flagging the address range 1000-9000 for prefetching. Then, a *Read* within this range triggers a predefined number of reads at following addresses. As these reads are generated from a point closer to off-chip memory than the CPU, the latency is reduced, and by the time the CPU requires the data being read, that data will already be in the CPU's data cache.

We have developed a simple implementation of such a smart router which is able to generate exactly one *Read* packet in the manner described above. It is extensible to provide more complex features of this sort.

2.7 Other Sorts of Routers

There are some plans to create further types of router for use within the memory access tree. The following routers will all have the same external interface as the simpler variety, but will add new features such as:

- *Level 2 caches*. Routers may serve some memory accesses from internal RAM, in which case they effectively act as level 2 caches.
- *Tracing*. Routers may log memory accesses to assist debugging.
- *Profiling*. Routers may record memory access patterns in order to create a profile of application behavior to assist performance improvements.

2.8 The Root of the Tree

The node at the root of the tree provides access to off-chip memory. In most of our prototype designs, this has been implemented as a bridge to an *Advanced Extensible Interface* (AXI) bus connection [22]. As illustrated in Figure 10, this allows any AXI-compatible memory controller to be connected to the tree, which includes all of the memory controllers for recent Xilinx FPGAs, e.g. [24].

The bridge interface is responsible for enforcing in-order processing of packets received via the tree. AXI allows read and write requests to be handled separately, so an arbiter is used to prevent ordering hazards when the bridge is processing multiple requests of mixed type.

In some other prototype designs, the root node is connected to internal FPGA memory instead of AXI. This is useful for simulation and debugging, though the internal memory is too small for storing anything beyond small test programs. The example highlights the fact that the root node can, in principle, be bridged to any sort of memory bus. For instance, *device transaction level* (DTL) protocol [15] links are a possibility. This will support the T-CREST off-chip memory controller described in section 3.

2.9 Status

The CPU tile and tree structure are now in fully working order on a Xilinx ML605 FPGA [23]. We have executed various test applications including the MRTC benchmark collection [12] and ported a real-time operating system to the platform [20]. The tests have validated the behavior of the caches, the SPMs and the shared memory tree.

For example, a 3×3 CPU grid has been synthesized for the Xilinx ML605 FPGA. All 9 CPUs are connected to the XPortMC shared memory tree. Software tests have verified that all 9 CPUs can receive messages from the others, and that all 9 can execute instructions and access data from the off-chip memory. The logic utilization for this system is reported by the Xilinx tools as:

```
Number of Slice Registers:          56124  out of  301440    18%
Number of Slice LUTs:               70089  out of  150720    46%
Number of Block RAM/FIFO:              87  out of     416    20%
```

A total of 512Mb of off-chip memory is available and working, and initial tests suggest that a 4×4 grid is quite possible on the same FPGA device. The NoCs currently operate at 100MHz, but higher speeds are possible for some components (e.g. routers close to the external memory controller).

3 Off-Chip Memory Architecture Implementation

This section discusses reconfiguration of the off-chip SDRAM memory controller. Parts of the material is reused from D4.1 to explain terminology and ensure that this document is self-contained. Important differences with respect to the material in D4.1 are pointed out where applicable. This section is structured as follows. First in Section 3.1, we discuss the concept of use-cases and identify two important reconfiguration operations for the off-chip memory and explain their associated benefits. Furthermore, three reconfiguration service classes for real-time systems considered in this project are introduced. We then proceed in Section 3.2 by providing background on SDRAM memories, the baseline real-time memory controller used in the project, and the latency-rate abstraction, which are required for the technical discussion later in the document. Section 3.3 presents the architecture of the implemented reconfigurable real-time memory controller and a description of three particular architecture instances used in this project and their supported reconfiguration operations. In Section 3.4, we present a reconfiguration-friendly method to achieve predictable and composable behavior in a memory controller, based on composable memory patterns and TDM arbitration, and we discuss its performance implications. Section 3.5 then introduces a method for predictable and composable reconfiguration of the TDM arbiter to support use-case transitions. An overview of the implementation effort in the design-time and run-time flows of the memory controller is given in Section 3.6, before we conclude with an experimental evaluation of the proposed reconfigurable memory controller in Section 3.7.

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 11 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* [13] and are considered one of the main challenges of this project. As an example, the MP3 playback application in Figure 11 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.

This project considers two 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, response times, and energy consumption. Efficiently satisfying these constraints requires the memory controller to be reconfigured to enable different trade-offs between these properties. This trade-off can be covered by the second reconfig-

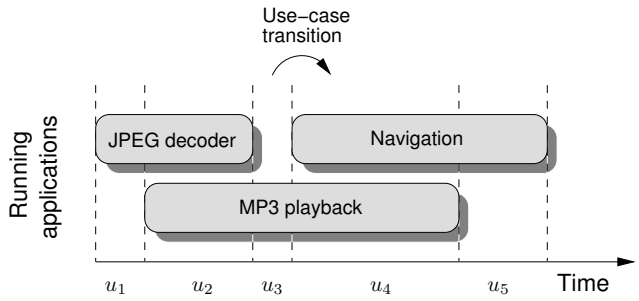


Figure 11: Starting and stopping applications triggers use-case transitions.

uration option, which 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 execution times [11] and energy consumption (although power consumption increases). Note that frequency scaling of the memory was considered as another driver for reconfiguration in D4.2, but after further investigation this turned out to be too slow to be practical for use-case transitions in real-time systems and the benefits in terms of energy consumption would be limited compared to existing power-down policies. Frequency scaling will hence not be discussed further in this document.

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 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. All three have been implemented in both a SystemC simulation environment and in VHDL for execution in hardware. The three classes are listed below in order of ascending difficulty:

1. *Reconfiguration without persistent service* just requires the ability to reconfigure the memory controller 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 *actual response times* of memory requests from persistent memory clients are not changed by a single clock cycle during or after reconfiguration in addition to functional correctness.

3.2 Background

3.2.1 SDRAM Specifics

SDRAM memories are challenging to use in systems with real-time requirements because of their internal architecture. An SDRAM memory comprises a number of banks, each containing a memory

array with a matrix-like structure, consisting of rows and columns [14]. Each bank has a row buffer that can hold one open row at a time, and read and write operations are only allowed to the open row.

The behavior of an SDRAM memory is determined by the sequence of SDRAM commands that are communicated from the memory controller to the memory device. These commands tell the memory to open a particular row in the memory array, to read or write a burst to/from an open row, or to close an open row and store its contents back into the memory array. There is also a *refresh* command that charges the capacitors of the memory elements to ensure that the contents of the memory array are retained. Scheduling SDRAM commands is not a trivial task, since there is a considerable number of timing constraints that must be satisfied before a command can be issued. These timing constraints are minimum delays between issuing particular SDRAM commands.

The SDRAM architecture makes the execution time of requests highly variable for three reasons. 1) A request targeting an open row can be served immediately, while it otherwise first needs the current row to be closed and the required row to be opened. 2) The data bus is bi-directional and requires several cycles to switch from read to write and vice versa. 3) The memory must occasionally be refreshed before executing the next request. The impact of these factors may cause the execution time of an SDRAM burst to vary by an order of magnitude from a few clock cycles to a few tens of cycles.

3.2.2 Real-Time SDRAM Controller

The work in this document is based on the hybrid real-time memory controller presented in [2]. The hybrid concept is based on *predictable memory patterns*, which are statically computed sequences (sub-schedules) of SDRAM commands. These patterns are dynamically executed in a non-preemptive manner at run-time based on incoming requests. The memory patterns exist in six flavors: 1) read pattern (R), 2) write pattern (W), 3) read/write switching pattern (RtW), 4) write/read switching pattern (WtR), 5) refresh pattern (REF), and 6) idle pattern (I). The read and write patterns are referred to as *access patterns* and they access the memory with a certain *access granularity* known as an atomic service unit, also referred to as *atoms*. *Switching patterns* execute either before an upcoming read or write pattern if the previously executing access pattern was of the other type. An example of how requests map to memory patterns is shown in Figure 12.

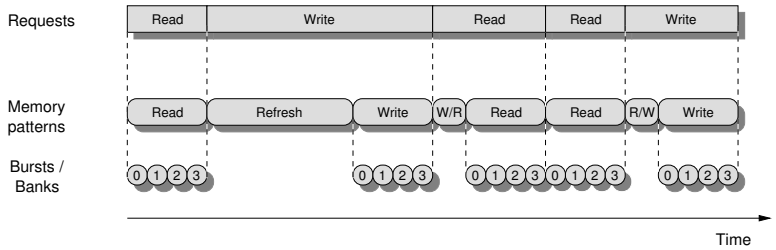


Figure 12: Mapping from requests to memory patterns to SDRAM bursts.

The access patterns have a certain *bank interleaving* (BI) and *burst count* (BC) number to set the degree of bank-level parallelism and the number of read or write bursts within a pattern. The combinations of BIs and BCs offer different trade-offs between bandwidth, response time, and energy, and are selected on the requirements of the clients [11]. The patterns are *automatically generated* [1]

at design time based on the desired trade-off and the timing constraints of the particular SDRAM device. Patterns hence access a known amount of data within a bounded amount of time, providing bounds on bandwidth and execution times [5].

Gross bandwidth defines the minimum guaranteed bandwidth the SDRAM can offer. Applications are guaranteed a certain *net bandwidth*, which is a fraction of the gross bandwidth. Gross bandwidth is the bandwidth pattern sets offer during back-to-back execution of worst-case ordered patterns [5]. Pattern sets are *read-dominant* or *write-dominant* when the respective access pattern is longer than the combination of the other access pattern and both switching patterns. For these pattern sets, worst-case bandwidth occurs when the dominating pattern is used continuously. Pattern sets are *mixed-dominant* when either combination of switching pattern and following access pattern is longest. These pattern sets offer minimum bandwidth when read and write requests interleave continuously, maximizing the number of switches.

3.2.3 Latency-Rate Servers

SDRAM performance bounds in the memory controller are based on the Latency-Rate ($\mathcal{LR}$) server model [19]. They are able to abstract shared resource behavior. $\mathcal{LR}$ servers guarantee memory clients a minimum *allocated rate* of service after a maximum *service latency*, shown graphically in Figure 13. This linear service guarantee bounds the amount of data that can be transferred during any interval independently of the behavior of other clients. The allocated rate and service latency depend on the particular arbiter and its configuration.

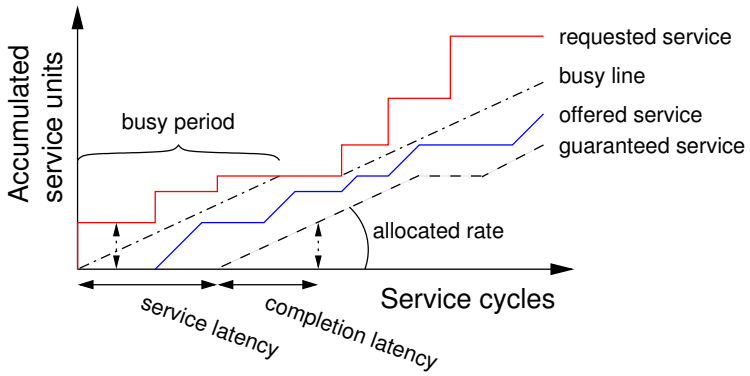


Figure 13: A $\mathcal{LR}$ server and its associated concepts.

The $\mathcal{LR}$ service guarantee is conditional and only applies if the master produces enough requests to keep the server busy. This is captured by the concept of *busy periods*, which intuitively are periods in which a client requests at least as much service as it has been allocated on average. This is illustrated in Figure 13, where the client is busy when the requested service curve is above the dash-dotted reference line with slope equal to the allocated rate that we informally refer to as the *busy line*. The figure also shows how the service bound is shifted when the master is not busy.

The real-time SDRAM controller is abstracted as a $\mathcal{LR}$ server and its $\mathcal{LR}$ guarantees allow for derivation of *worst-case response times* (WCRT) of all requests. For the first request of busy periods,

worst-case delay from arrival to receiving service is bounded by the service latency. Following requests receive service no later than the worst-case *finishing time* of the previous request. A request finishing time is bounded by taking its worst-case delay to service and adding the time required to service the request at the allocated rate, referred to as the *completion latency*.

In this document, the basic $\mathcal{LR}$ model is slightly modified to optimize performance for predictable SDRAM [17]. Completion latencies are generally determined using the allocated rate and assuming a fully preemptive resource. The optimization exploits that the memory controller is only preemptive on the granularity of patterns, which enables lower response times to be calculated.

3.3 Architecture of the Reconfigurable Controller

This section introduces the architecture of the 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, response times, and energy consumption for each use-case. The architecture, shown in Figure 14, 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.

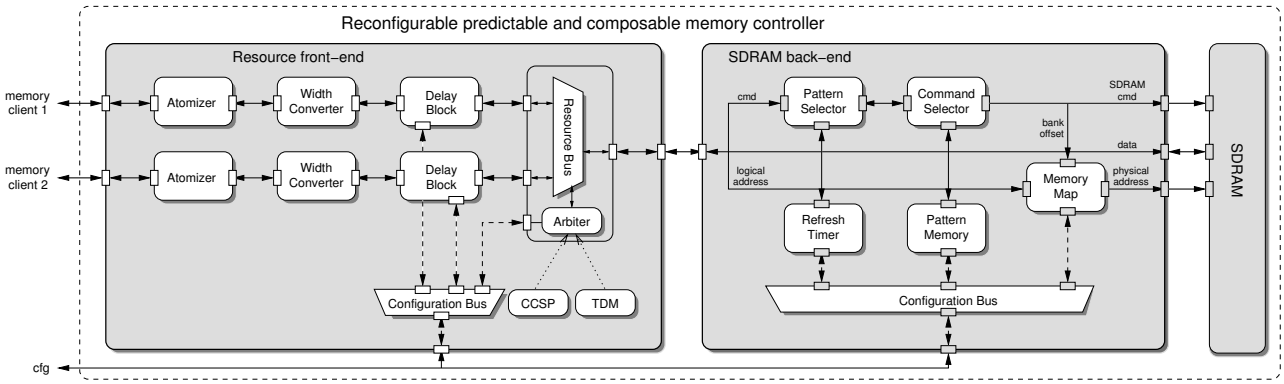


Figure 14: Architecture of reconfigurable predictable and composable memory controller.

3.3.1 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 to be programmed via memory mapped I/O during use-case transitions. The blocks communicate using a *device transaction level* (DTL) protocol [15], which is a standardized communication protocol similar to AXI. All white ports shown in Figure 14 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 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 depends on the chosen BI/BC of the memory patterns. Reconfiguring the access granularity of the memory enables a different trade-off between bandwidth, response time, and energy.

The Width Converter is used to convert from the word width used by the rest of the system to the width used by the memory controller. In our case, this implies a conversion from a word width of 32 bits to 128 bits. The memory device itself has a usable width of 32 bits, but runs at twice the frequency of the memory controller and transfers two words per clock cycle due to the double data rate of DDR memories. The conversion is done by simply buffering four words arriving sequentially and issue them in parallel towards the next block. Similarly, in the other direction, 128 bit words are serialized and passed on.

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 (and thus the entire shared resource) 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) [7]. The configuration of the arbiter is use-case dependent and needs to be reprogrammed on use-case transitions. The infrastructure and protocol to reconfigure the arbiter in a predictable and composable manner was developed as a part of this project.

3.3.2 SDRAM Back-End

Next we discuss the architecture of the reconfigurable SDRAM back-end that has been developed in this project. At high level, the back-end comprises five main functional blocks, the *Pattern Selector*, *Command Selector*, *Refresh Timer*, *Pattern Memory*, and the *Memory Map*.

The Pattern Selector looks at the command (type) of the incoming request and determines if it is a read or a write. It then determines which memory pattern to schedule based on the type of the current and previous requests and whether or not it is time to refresh the memory, the latter being indicated by the Refresh Timer.

Having decided on what pattern to schedule, a pattern identifier is sent to the Command Selector that looks up the pattern in the Pattern Memory and issues two SDRAM commands per clock cycle (since

the memory runs at twice the frequency) until the pattern is completed. The Pattern Memory stores the command and a relative bank address indicating which among the banks it interleaves over it is targeting, e.g. one or two if BI=2. This relative address is required for address generation.

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 [14]. Once the physical base address of the request is computed, the Memory Map generates addresses for every command and passes it on to the memory device.

The three configurable blocks in the back-end are the Refresh Timer, the Pattern Memory, and the Memory Map, all currently configured via an FSL interface. These blocks are hence connected to the Configuration Bus in the back-end. The Refresh Timer and Pattern Memory are configured to accommodate different memory devices with different frequencies, thus requiring different patterns and refresh frequencies. The Memory Map can be programmed to translate addresses 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 energy consumption, since more memory banks have to be opened and closed per request [10]. The Memory Map is programmable to allow trade-offs between worst-case and average-case bandwidth, response times, and energy consumption to be made. 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.

3.3.3 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 controller back-end is a composable resource or just predictable [6]. The implemented reconfigurable controller 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 energy consumption and response times. 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 reservations 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 [13]. 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 [6]. 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. This instance has been implemented both as a SystemC simulation model and in VHDL and is extensively discussed and evaluated throughout this document.

Table 1: Theoretically and actually (bold) supported reconfiguration options for the different architecture instances.

Reconfiguration class	No persistence		Persistent pred.		Persistent comp.	
Operation	Start	AG	Start	AG	Start	AG
TDM comp. patterns	**yes**	**yes**	**yes**	no	**yes**	no
TDM & Delay Block	**yes**	**yes**	**yes**	no	**yes**	no
CCSP & Delay Block	**yes**	**yes**	yes	no	yes	no

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. This instance has been implemented both as a SystemC simulation model and in VHDL.

3. *Any predictable arbiter with predictable memory and Delay Block.* The most general and complex architecture considered for the memory controller. Instances with different predictable arbiters, such as CCSP [7], and Delay Blocks have been implemented in both SystemC and VHDL. However, the transition behavior of each arbiter needs to be examined and a reconfiguration protocol must be derived to safely reconfigure in bounded time. Such protocol has only been derived for TDM arbitration due to limited time. This protocol is presented in Section 3.5.

Table 1 concludes this section by summarizing which reconfiguration options are theoretically supported for the different architecture instances and reconfiguration service classes. The implemented options are highlighted in bold. The table covers two reconfiguration operations for every combination of reconfiguration class and architecture instance: 1) starting and stopping memory clients (start), and 2) changing the access granularity of the memory controller (AG). Starting and stopping memory clients is supported in all instances using TDM arbitration, but not for general predictable arbiters such as CCSP. In contrast to what was mentioned in D4.1, it is not possible to reconfigure the access granularity on use-case transitions with persistent memory clients. The reason is that this involves changing the memory map and thus that the logical addresses of data belonging to persistent clients changes, making them unable to read data at the address they wrote it.

3.4 Approach to Real-Time SDRAM access

This section presents a reconfiguration-friendly method to achieve predictable and composable SDRAM access based on combining composable memory patterns with TDM arbitration. First, we introduce TDM arbitration and discuss its real-time properties. We then proceed by presenting composable memory patterns before concluding with a brief discussion about the performance implications of the approach.

3.4.1 TDM Arbitration

TDM arbiters regulate access to a resource across multiple clients by dividing time into *frames* comprising a number of *slots*. Slots are allocated to clients, each corresponding to non-preemptive ser-

vice of an atom. A *scheduling decision* is made every time a request is serviced or, if idle, when the *schedule interval* has passed. This interval is equal to the execution time of the shortest access pattern. Every scheduling decision the *index* is updated, selecting the next slot for scheduling. A new *frame iteration* is started if the index moves from the last to the first slot.

A TDM configuration consists of 1) a frame size parameter which determines the number of slots in the frame, and 2) an allocation of slots to clients. Many allocation strategies exist, but these are not the focus of this work. A *greedy allocation* strategy is used, which allocates continuous blocks of slots per application. TDM is a predictable arbitration mechanism and a bound on the number of interfering requests using the greedy allocation strategy has been previously presented in [6].

Conceptually, TDM arbiters are composable by design. Clients always receive service for their allocated duration, independently of who is serviced for the remainder of the frame iteration. However, its composability property depends on what the arbiter decides to do when a scheduled client does not have any pending requests. A *work-conserving* arbiter schedules another client with pending requests. A *non-work-conserving* arbiter does not schedule any and the resource falls idle. For a work-conserving TDM arbiter, the instant when requests for a particular client receive service is dependent on the number of pending requests of other clients. This dependence between memory clients may cause requests to be served sooner than expected, making the resource non-composable. Non-work-conserving arbiters do not have this dependence and are conceptually composable.

Using a non-work-conserving TDM arbiter to arbitrate an SDRAM does not immediately give composable behavior. The reason is varying slot sizes. Slot sizes vary because execution times of atoms do, and they cannot be preempted. Execution times are variable for three reasons: 1) Read and write requests may have a different execution time, depending on the corresponding patterns lengths. 2) Switching patterns are required to execute before the access pattern when a read request follows a write request or vice versa. 3) A scheduled memory client may experience delayed execution due to a refresh. Occurrence of the first and second reason is shown in Figure 15. The figure shows execution of four consecutive requests, all with different execution times. Only the first two reasons result in interference between memory clients. Firstly, different access pattern lengths cause time instants of upcoming scheduling decisions to vary depending on the current request type, and consequently affecting the scheduling time of future request. Secondly, the execution time of the current request is dependent on the type of the previously serviced request, via a potentially required switching pattern. The third reason, interference from refreshes, can be ignored because it is not induced by other memory clients.

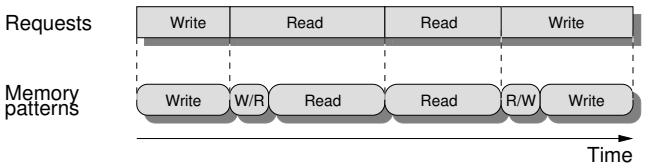


Figure 15: Predictable pattern execution

3.4.2 Composable Patterns

Composable patterns are proposed to solve slot size dependence on other memory clients. Composable patterns are memory patterns with two properties that overcome the non-composable nature of the original *predictable patterns*. First, all access patterns are equally long and second, switching patterns are always executed by incorporating them into the access patterns. The same execution trace as shown in Figure 15 is shown in Figure 16 using composable patterns. The execution times of all requests are equal and consequently slot sizes are too.

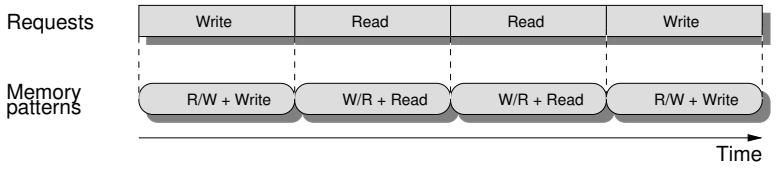


Figure 16: Composable pattern execution

Composable patterns are generated by adapting predictable patterns, still considering the original timing constraints. Two different methods are used, depending on the pattern set dominance property. For mixed-dominant patterns: First, switching patterns are inserted in full in front of their corresponding access pattern, i.e. the read-to-write switching pattern is inserted before the write pattern, and the write-to-read switching pattern is added before the read pattern. Second, after appending the switching patterns, the access patterns may have different lengths. SDRAM idle commands (NOP) are appended to the shortest access pattern to make them equally long. For read-dominant or write-dominant pattern sets, there are no switches in the worst case. In this case, the shorter access pattern is appended with NOPs to give it equal length as the longer pattern. Equation (1) describes the relation between access patterns (AP) lengths of a composable pattern set P_c and a predictable pattern set P_p, for both mixed-dominant *(md)* and read-dominant or write-dominant *(rdwd)* pattern sets.

$$AP_c = \begin{cases} max(R_p, W_p) & \textit{(rdwd)} \\ max(WtR_p + R_p, RtW_p + W_p) & \textit{(md)} \end{cases} \quad (1)$$

An example result of converting predictable patterns to composable patterns is shown in Table 2. Besides the newly sized access patterns and removed switching patterns, also the refresh pattern and introduced idle pattern (I) are shown. Refreshes do not affect composability and hence the refresh pattern can be sized differently in the two pattern sets.

We proceed by briefly discussing the performance implications of using composable patterns. Composable access patterns are on average at least as long, but often longer than the original predictable patterns. The same amount of data is accessed using longer patterns, implying composable patterns have reduced efficiency and offer reduced gross bandwidth. Use-cases with high bandwidth requirements may not be allocated successfully using composable patterns, but would have been with predictable patterns. Equation (2) analytically expresses the relation between gross bandwidth

Table 2: Pattern lengths (mixed-dominant set)

	R	W	WtR	RtW	REF	I
P_p	31	35	5	3	44	31
P_c	38	38	0	0	44	38

for predictable and composable patterns using the efficiency factor e^{pc}, which indicates the fraction of the guaranteed gross bandwidth offered by predictable patterns that are provided by composable patterns. Gross bandwidth is not affected for read-dominant or write-dominant pattern sets, because the dominant pattern has not changed.

$$e^{\text{pc}} = \begin{cases} 1 & \textit{(rdwd)} \\ \frac{R_p+W_p+WtR_p+RtW_p}{R_c+W_c} & \textit{(md)} \end{cases} \tag{2}$$

Composable patterns also have an effect on response times of requests, due to the potentially increased execution times. The impact of this is difficult to determine since it depends on the number of required switching patterns, which is arbiter, use-case, and application dependent. However, a lower bound on this impact can be determined by taking pattern lengths into account and considering a worst-case ordered request stream. A lower bound on the increase in response time is then given by $1/e^{\text{pc}}$. An experimental evaluation of the composable patterns is later presented in Section 3.7.

3.5 Reconfigurable TDM Arbitration

We have presented the concept of composable memory patterns that turn the memory controller back-end into a predictable and composable resource and we briefly discussed their performance implications in terms of guaranteed bandwidth and response times. We now proceed by discussing the implications of sharing this resource in a system with multiple use-cases. Each use-case has a unique set of requirements and upon a use-case switch, the system must be able to stop and start applications and reconfigure affected resources such that they satisfy the updated requirements. Such a reconfiguration is a system-wide process. Events take place on three levels: 1) Reconfiguration on system-level, i.e. use-case switching, initiated by starting and stopping applications. 2) Reconfiguration of applications, i.e. changes to the configuration of applications during their lifetime. 3) Arbiter reconfiguration, the process of making changes to the TDM frame at run-time. Slots are reconfigured to match the configuration of starting use-cases, as determined at design time.

Start and stop reconfiguration events are not challenging themselves. Applications may require $\mathcal{LR}$ guarantees and composability for verification, but only during their *lifetime*, the period in which they are *active*. By definition, start and stop events take place outside application lifetime. Starting and stopping applications, on arbiter level adding and removal of their allocated slots, can be done arbitrarily. The challenge for use-case switching is how other applications, besides the one inducing the use-case switch, experience the reconfiguration. Different types of applications are distinguished,

based on their real-time requirements during use-case switches. All applications receive predictable and composable service during regular execution. *Persistent composable applications* are real-time applications and active in multiple use-cases, requiring composability for independent verification by simulation. The challenge is to process reconfigurations for use-case switching without affecting their timing behavior, thus losing composability. *Persistent predictable applications* are real-time applications which depend on $\mathcal{LR}$ guarantees for design-time verification using formal performance analysis. These applications can be affected by reconfiguration events, as long as their $\mathcal{LR}$ guarantees are not invalidated. *Non-persistent applications* are active in only one use-case and never experience arbiter reconfiguration. In this section, we address these challenges by proposing a method of predictable and composable TDM arbiter reconfiguration on use-case switches with persistent clients.

3.5.1 TDM Slot Allocation

Use-case switching with persistent composable applications has two prerequisites. 1) The reconfiguration process needs to be independent of the scheduling process, such that processing reconfiguration events does not delay scheduling decisions. This is easily satisfied when scheduling and reconfiguration are handled by separate sub-modules of the arbiter. 2) Persistent composable applications are never reconfigured during their lifetime. They must have identical slot allocations over all their use-cases. This complicates the allocation process due to coupling in configurations between use-cases. The actual effects depend on the used allocation strategy. For this work, using greedy allocation, fragmentation of the TDM frame becomes an issue.

Algorithm 1 determines slot allocations. Persistent composable applications are given priority and receive allocation over all their use-cases. Next, all persistent predictable applications are allocated to their use-cases. Persistent predictable applications may have different allocations over their use-cases, due to already allocated persistent composable applications, and may require reconfiguration during their lifetime. This has to be done without invalidating their $\mathcal{LR}$ guarantees. Lastly, non-persistent applications receive their allocation.

3.5.2 Predictable TDM Reconfiguration

Persistent predictable applications require reconfiguration if their slot allocations differ between the two use-cases. If so, allocated slots are *moved* by adding newly allocated slots and removing old ones. Behavior for both configurations has been verified at design time. However, their behavior is temporarily undetermined during the use-case switch when slots are added and removed. This is only allowed if behavior stays predictable, i.e. still provides the $\mathcal{LR}$ guarantees. Two prerequisites have to be satisfied: 1) An application has to receive service for at least the allocated number of slots during each frame iteration. Failure to satisfy this temporarily decreases the allocated rate and invalidates completion latencies. 2) *Arbiter latencies*, the maximum interval between two slots of an allocation, cannot exceed the value determined at design time for regular execution. This would invalidate the service latency bound.

Figure 17 shows two reconfiguration scenarios. An application is allocated one slot in a frame of four. The index moves right as time passes, every fourth update selecting the first slot again. Arrow lengths indicate observed arbiter latency, expressed in slots, being three during regular execution. A

Algorithm 1 Allocation algorithm

Input: persistent composable applications $\vec{A_c}$, other applications $\vec{A_o}$,
slot requirements $\vec{R}$, use-cases $\vec{U}$,
Output: TDM configurations $\vec{C}$

```
for all a in A_c do
    location ← 0
    for all u in U[a] do
        repeat
            S[u] ← findSpace(location, R[a])
            location ← increment(location)
        until S[u] or reached end of frame
    end for
    if S = 1 then
        C[U[a]] ← setSlots(a, location, R[a])
    end if
end for
for all a in A_o do
    location ← 0
    for all u in U[a] do
        repeat
            S ← findSpace(location, R[a])
            location ← increment(location)
        until S or reached end of frame
        if S then
            C[u] ← setSlots(a, location, R[a])
        end if
    end for
end for
```

reconfiguration event that moves the allocated slot takes place at time t_r when the index is at the appointed location. Scenario $1a$ and $2a$ add and remove slots directly and simultaneously at t_r. For Scenario 1, the allocated slot moves to the left. For $1a$, due to the unfortunate reconfiguration time instant, the allocated slot moves past the index. A full frame iteration without service occurs, both increasing arbiter latency beyond the regular value and temporarily reducing the allocated rate to zero. For Scenario 2, the allocated slot moves to the right. For $2a$, arbiter latency again increases beyond its regular value, invalidating the service latency.

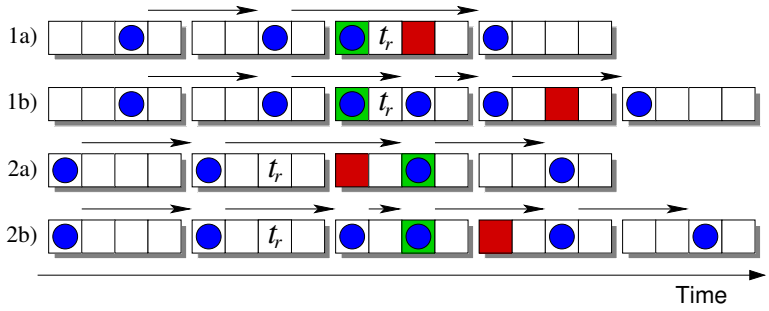


Figure 17: Arbiter latencies during reconfiguration. Unpredictable: $1a$ and $2a$. Predictable: $1b$ and $2b$

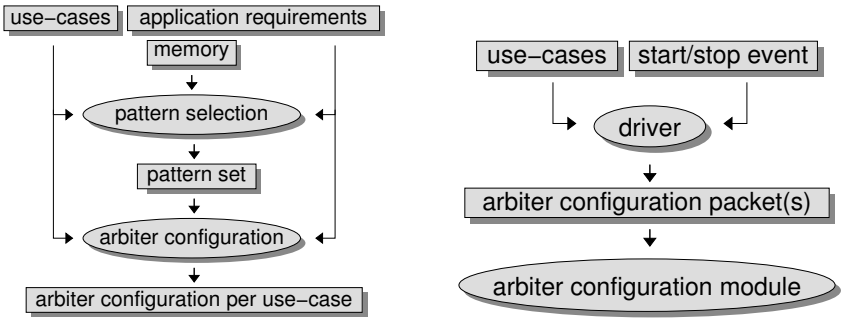
It is possible to reconfigure the TDM arbiter without invalidating guarantees for the reconfigured application. The solution consists of three steps. 1) The reconfiguration event for moving slots consists of two distinct reconfiguration actions to add and remove slots. Of these two actions, adding is always processed before removing. 2) Processing a split reconfiguration event is never interrupted by events related to other applications. 3) Removal of slots is only allowed at the beginning of a frame, i.e. at the start of a new frame iteration. These rules assure that adding and removing slots is done in separate frame iterations and slots do not get added to another client before being properly removed. Reconfiguration using this method is shown in Figure 17, scenarios $1b$ and $2b$. Arbiter latencies do not increase and frame iterations with too few allocated slots do not occur. For a single frame iteration, moved applications temporarily have more slots allocated to them than during regular execution. Response times temporarily decrease and offered service increases, temporarily improving application performance. Both service latency and completion latency are valid during reconfiguration and $\mathcal{LR}$ guarantees are satisfied.

It is worth noting that it is possible to offer composability to persistent predictable applications, which may be reconfigured. $\mathcal{LR}$ guarantees are valid during reconfiguration and Delay Blocks can delay to a valid WCRT. The proposed arbiter reconfiguration method is particularly appealing, because WCRT do not increase. Alternatively to the proposed method, one could also determine WCRT during reconfiguration and apply these to regular execution, although this results in higher WCRT.

3.6 Implementation

This section describes the implementation and integration of composable patterns and a reconfigurable TDM arbiter with supporting functionality. A composable SDRAM controller and use-case

switching capabilities require support on multiple levels, as shown in Figure 18. Top-down, there are: 1) At design time, composable pattern generation and support for the tool flow to consider multiple use-cases to determine run-time configurations per use-case 2) At run-time, support for reconfiguration in the arbiter driver software. 3) In hardware, the arbiter module itself and an infrastructure to deliver reconfiguration events at the arbiter. The following subsections elaborate on each of them.



(a) Design-time flow (b) Run-time flow

Figure 18: Design-time and run-time flows.

3.6.1 Design-Time Tool Flow

The tool flow determines an arbiter configuration at design time using the flow shown in Figure 18a. The flow is identical with or without support for multiple use-cases, except for the use-case data. The tool flow previously used a super-use-case, the union of all use-cases, for configuration of the SDRAM controller. With reconfiguration support, it uses *maximum cliques* of simultaneously running applications. All cliques part of a maximum clique share a configuration, significantly reducing the number of configurations. The tool flow output is now multiple arbiter configurations instead of just one, all using a common pattern set.

In the pattern generation and selection process, support is added in two areas. First, there is the pattern generation, to which conversion to composable patterns is added. Second, the pattern selection process is to consider all use-cases instead of the previously used super-use-case. With the ability to switch use-cases, all use-cases are individually considered to determine maximum *slack bandwidth*. The selected pattern set offers most slack summed over all use-cases. Applications that are in many use-cases hence influence final pattern selection more than those who are in few.

Arbiter configuration takes place as discussed in Section 3.5 and shown in Algorithm 1. At this stage of the tool flow, application requirements are expressed in number of slots. The selected pattern set is used to convert bandwidth requirements to slot requirements, as explained in [6].

3.6.2 Run-Time Software

Support for multiple use-cases requires support in both software and hardware, as shown in Figure 18b. Upon a system-level application start or stop, the three steps involved are: 1) checking the need for reconfiguration, 2) creating configuration events, and 3) sorting reconfiguration events. These steps are described next.

Checking the Need for Reconfiguration Not all use-case switches require reconfiguration for other applications besides the one starting or stopping. Use-case switches within the same maximum clique share configurations and reconfiguration is thus not necessary. If the upcoming use-case is part of a different maximum clique reconfiguration may be required and memory clients that are in both the current and next use-case are marked.

Creating Reconfiguration Events The driver compiles a list of reconfiguration events. Such events contain information about which slots are to be added or removed for a client. For marked memory clients, allocations on both sides of the use-case switch are compared and a reconfiguration event is created if they differ to move their slots. Such a move comprises separate adding and removal actions. Overlap between allocations of the current and next configuration is handled when creating the removal action. Recently added slots, by the adding action of the same reconfiguration event, should not be removed immediately afterward by the following removal action. Without this precaution, overlapping slots between the two configurations are missing from the new allocation and the resulting reduced allocated rate leads to violation of the $\mathcal{LR}$ guarantees. Reconfiguration events for starting or stopping applications are straightforward.

Sorting Reconfiguration Events The reconfiguration event list is sorted to determine in which order they should be performed. The reconfiguration order of active memory clients matters, because the current and next allocations of different clients may interfere, i.e. the allocation for an application in the next use-case that is reconfigured first, uses slots still allocated to another application in the current use-case. Occurrence of such a situation is the switching condition for an algorithm to order all reconfiguration events such that allocations do not interfere. The worst-case computational complexity $O(n^2)$ is not an issue for a configuration host for any practical number of applications. Circular dependencies are conceptually possible if two applications switch slots, but do not occur with the used allocation strategy.

3.6.3 Run-Time Hardware

Hardware related to TDM reconfiguration support is limited to the infrastructure used to deliver reconfiguration events at the arbiter, and the arbiter itself. The TDM arbiter is implemented as discussed in Section 3.5. Scheduling and reconfiguration processes are implemented as distinct sub-modules, such that they work in parallel. Furthermore, a *shadow frame* is used. This frame is a second TDM frame, not used for making scheduling decisions, but only for reconfiguration purposes. Shadow frame contents are copied into the regular frame at transitions between frame iterations, making reconfigurations effective only at this time. Removal of slots is only allowed during a transition between frame iterations and adding slots is only allowed directly if no removal event is waiting. Adding events that can be processed directly are processed on the regular frame. Removal and queued adding of slots are performed on the shadow frame.

3.7 Experimental Results

This section experimentally evaluates the proposed reconfigurable memory controller. First, the two experimental setups are described before we proceed by evaluating the performance impact of composable patterns. Lastly, we conclude by experimentally demonstrating that the memory controller provides predictable and composable behavior during reconfiguration.

3.7.1 Experimental Setup

Experiments are performed both on the SystemC simulation environment and hardware platform on FPGA. A Xilinx ML605 [23] is used, equipped with a Micron MT4JSF6464HY 512MB DDR3 SDRAM SODIMM. A model of this SDRAM is available and used for the tool flow and simulations.

SystemC Simulation Model A SystemC implementation of the memory controller is used to access the SDRAM. Applications are implemented by either *traffic generators* or *trace players*. Traffic generators generate synthetic requests as specified by the application requirements, i.e. response times and read and write bandwidth. Variations in request frequency around the mean bandwidth are controllable. Alternatively to synthetically generated traffic, *traces* can be used. Trace players execute cycle-precise traces of real applications. Traces are generated by running applications on a *SimpleScalar* [8] ARM simulator.

FPGA Platform The hardware implementation uses Xilinx Microblaze CPUs execute applications [21]. For this work, no operating system is used such that they run only a single application at a time. The applications used here are synthetic applications, doing read or write requests with a certain bandwidth as specified. Like the traffic generators, the burstiness of the traffic can be specified.

3.7.2 Performance of Composable Patterns

As discussed in Section 3.4, the conversion from predictable to composable patterns may cause a gross bandwidth reduction. An experiment is performed to determine exact losses for a diverse range of SDRAM and pattern sets. Gross bandwidth for an SDRAM is known after pattern set generation, hence each time only the tool flow is run and no simulations or executions on FPGA are required.

Figure 19 shows gross bandwidth decrease with composable patterns for every supported BI and BC pattern set configuration for the Micron MT4JSF6464HY. Results are highly dependent on pattern configurations, ranging from 0% reduction for every pattern with BI1 to 7.9% for the pattern set with BI4 and BC2. It is out of scope of this work to elaborately discuss reasons for particular performance of each pattern configuration, as it mainly depends on the pattern generation algorithm. However, two observations are listed next. 1) The difference in efficiency between predictable and composable patterns, e^{pc}, is mostly dependent on switching pattern sizes, because most pattern sets are mixed-dominant with equally long access patterns. 2) Pattern sets with BI1 do not suffer reduced gross bandwidth. For these, predictable and composable pattern sets are identical.

Table 3 shows summarized gross bandwidth reductions for twelve SDRAMs, divided over multiple SDRAM types. For each memory type DDR2, DDR3, LPDDR and LPDDR2, the average and maximum gross bandwidth reduction across pattern configurations are given. Results show that for specific pattern and memory configurations, gross bandwidth reduction can go up to 12%. For all SDRAM types on average over all pattern sets less than 1.6% of gross bandwidth is lost. For the majority of combinations of pattern sets, targeted SDRAM and use-case requirements, allocation chances are thus hardly affected.

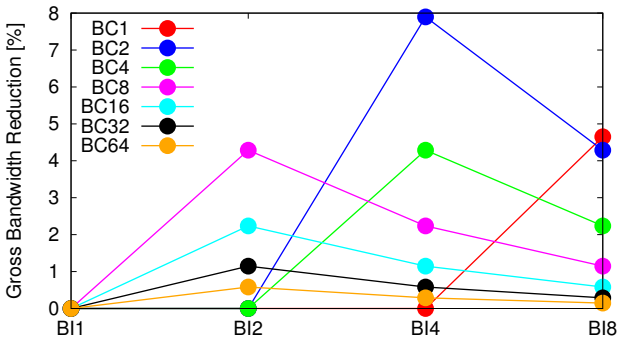


Figure 19: Gross bandwidth reduction per pattern set for MT4JSF6464HY

Table 3: Gross Bandwidth Reduction Summary

[%]	Average Gross Bandwidth Reduction	Maximum Gross Bandwidth Reduction
DDR2	1.3	9.5
DDR3	1.6	12.0
LPDDR	0.8	9.5
LPDDR2	0.3	5.0

3.7.3 Temporal Application Behavior

Predictable Reconfiguration A simulation experiment is set up with use-cases as shown in Figure 20. Applications $A - G$ are traffic generators (TG) with bandwidth requirements as given in Table 4. They are divided over use-cases $U1$, $U2$ and $U3$. Applications A and D are specified persistent composable and applications F and G specified persistent predictable. Each is active during at least one of the use-case switches $T1$ at 30 μs and $T2$ at 68 μs. Total execution time is 100 μs. Side-by-side applications share traffic generators, which is possible because they are mutually exclusive. All applications have a maximum service latency requirement of 2000 ns.

A pattern set with BI4 and BC1 is selected, offering a peak net bandwidth of 1862 MB/s. A super-use-case containing all applications requires 2880 MB/s and the experiment cannot be allocated successfully. Figure 21 shows the arbiter configuration for this experiment with use-case switching. The allocation is determined by Algorithm 1 with a set frame size of 20 slots. Persistent composable

Table 4: Bandwidth requirements for experiment in Figure 20

Application	A	B	C	D	E	F	G
Requested [MB/s]	320	240	200	320	800	600	400

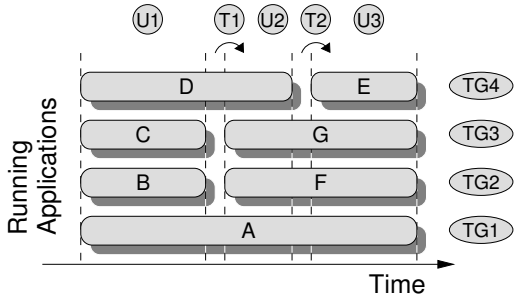


Figure 20: Use-cases for reconfiguration experiment

applications A and D have identical allocations over their use-cases, while persistent predictable applications F and G do not. Were they to have identical allocations between $U2$ and $U3$, application E would not be successfully allocated due to fragmentation.

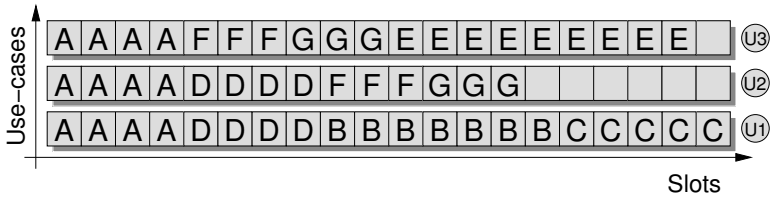


Figure 21: Arbiter configuration for experiment in Figure 20

Figure 22 shows finishing latencies for requests originating from $TG2$, which first runs application B and later F, with their WCRT as determined by their $\mathcal{LR}$ guarantees. The test is run twice, once with and once without the proposed method of predictable arbiter reconfiguration. Two reconfiguration events occur during the total run-time. The first event is visible at $T1$ at 30 μs, when application B is stopped and application F is started. The second event at $T2$ at 68 μs stops application D and starts application E, inducing a use-case switch from $U2$ to $U3$. Applications F and G are reconfigured, moving their allocations to accommodate application E. Figure 21 shows that both their allocations are moved left. The time instant for $T2$ is chosen to cause the situation shown in Figure 17 ($1a$) for application F.

Both runs show identical behavior during regular execution and when stopping and starting applications. Only when application F is reconfigured the behavior for F differs between runs. Arbitrarily adding and removing slots leads to invalidation of the $\mathcal{LR}$ guarantees, as some requests finish above their allowed WCRT. The proposed arbiter reconfiguration method maintains the $\mathcal{LR}$ guarantees.

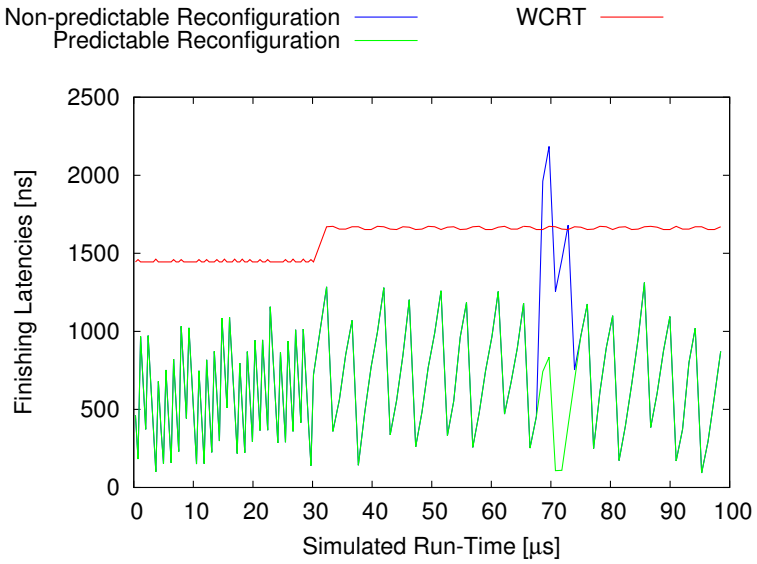


Figure 22: Reconfiguration at $T2$ for application F

$\mathcal{LR}$ Server Validation The experiment presented here investigates $\mathcal{LR}$ guarantees for the hardware instance on FPGA using composable patterns and TDM arbitration. The automated tool flow is used to instantiate and configure a platform with two Microblazes. Each Microblaze runs a single application doing 64 Byte SDRAM requests. *Application 1* (A1) only does read requests and *Application 2* (A2) only does write requests to ensure read/write interference. A2 requests a bandwidth of 80 MB/s, but periods of high and low request frequencies occur. During high request frequency intervals up to 180 MB/s is requested. A2 has a maximum allowed service latency of 1650 ns. A1 requests 300 MB/s with maximum allowed service latency of 2500 ns. Frame size is set to eight slots. With these parameters, the tool flow selects a composable pattern set with BI1 and BC2. A2 receives a single slot, guaranteeing 90 MB/s of bandwidth. A1 has four slots in the TDM frame.

Figure 23 illustrates accumulated requested service, provided service, guaranteed service and busy lines for A2. Only a small part of the total execution is shown to keep $\mathcal{LR}$ properties visible. The accumulated requested service increases with one atom for every request arrival and periods of high request frequencies initiate busy periods. Note that the provided service is always above the guaranteed service of 90 MB/s. The actual service latencies differ because of the differing arbiter state, such as the TDM index location, at request arrivals. We conclude that the service latency appears to be correctly bounded and service is provided at least according to the allocated bandwidth, experimentally validating the $\mathcal{LR}$ guarantee.

Composability The previous FPGA experiment is extended to show composable behavior for A1. Persistent composable application A1 performs 100 read requests at 360 MB/s and is run three times, each time under different circumstances. A) A2 is inactive and does not attempt to interfere. It has no allocated slots in the TDM frame. B) A2 is active, does write requests and has a bandwidth allocation of 270 MB/s, corresponding to four slots in the TDM frame. C) A2 is active and does writes. Its initial bandwidth allocation of 90 MB/s is reconfigured to 180 MB/s after 35 μs. Its allocation changes from

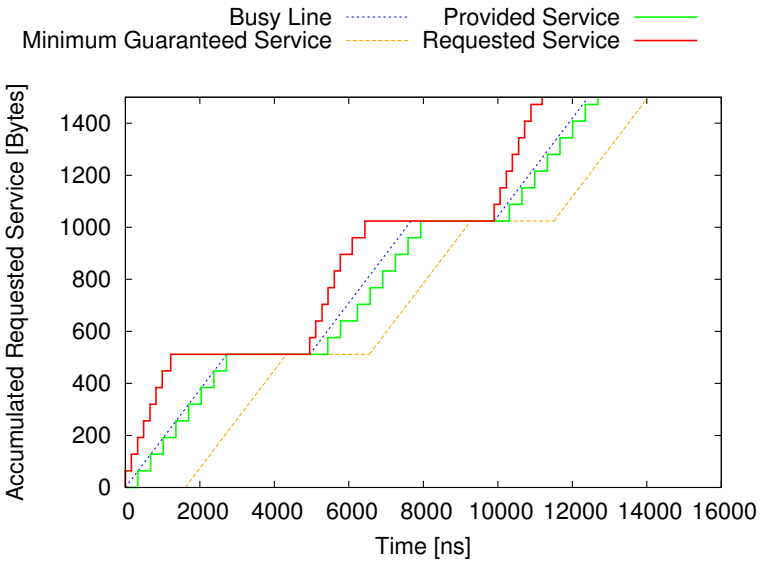


Figure 23: $\mathcal{LR}$ server behavior for SDRAM for FPGA implementation of the memory controller.

one to two slots. Furthermore for all cases, frame size is set to eight and service latency requirements are chosen such that again a pattern set with BI1 and BC2 is selected. The experiment is performed twice, once with predictable patterns and once with composable patterns.

Figure 24 shows temporal behavior of the applications when using predictable patterns. The green line shows behavior of A1 for A, when it is the only active application and no attempts to interfere are made. This is considered its reference behavior. Blue and red lines indicate behavior for A2 for cases B and C. For these cases, A1 behavior is shown respectively as exes and circles. Composability is experimentally demonstrated if A1 has identical behavior for A, B and C, i.e. if exes and circles are positioned on top of each other and on the green line. This is not the case. Interference from A2 causes most requests of A1 to finish later, and some sooner, than without interference. Run-times for A, B, and C are different due to the specific interference behavior.

Figure 25 shows results for the experiment when using composable patterns. A1 is not affected by interfering requests from A2 during B, and is not affected by reconfiguration of A2 during C. The green line, exes and circles are on identical positions. Vertically, individual request latencies are shown to be identical. Horizontally, request arrivals occur at the same time instants, resulting in equal run-times. Furthermore, identical behavior over multiple runs indicates that the platform is deterministic, although it is not required for composability. The results show that the reconfigurable SDRAM controller is a composable resource.

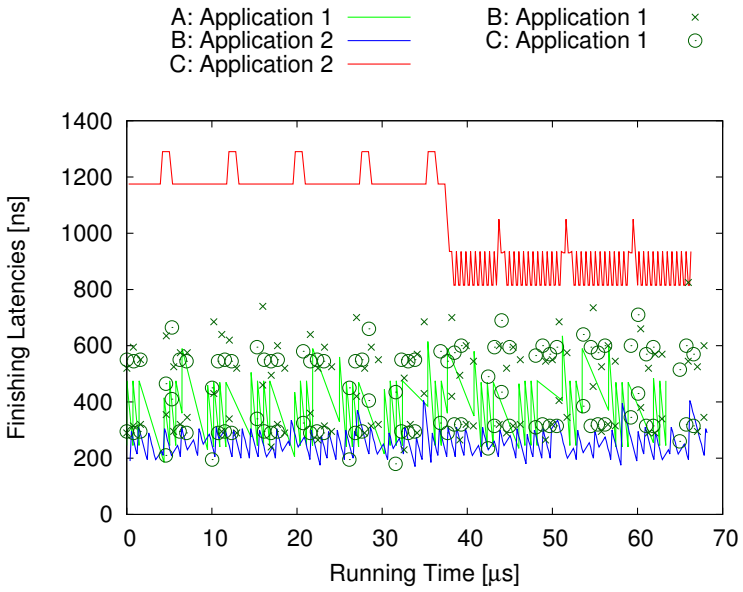


Figure 24: Non-composable behavior using predictable patterns on FPGA

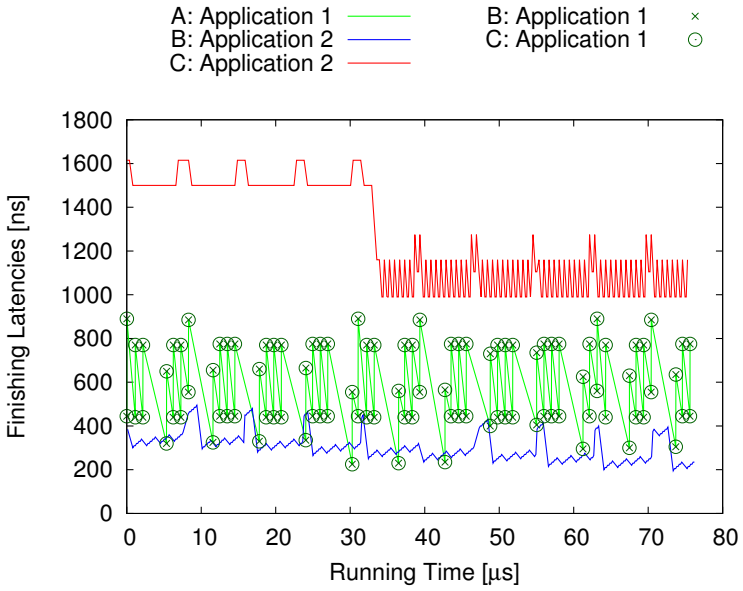


Figure 25: Composable behavior using composable patterns on FPGA

4 Requirements

This section lists all requirements in aspect CORE and scope NEAR from Deliverable D 1.1 that are relevant for the memory work package. NON-CORE and FAR requirements are not listed here. The requirements are followed by a comment regarding the extent to which it is fulfilled by the research concept in this document.

M-2-010 No Re-Order:

The processor may have several read or write requests outstanding. The memory controller shall not reorder these read or write requests from the processor.

The implemented reconfigurable memory controller preserves FIFO ordering between requests from its memory clients.

M-2-012 Compare-and-Swap:

The memory controller (and network interface) shall support a CAS (compare-and-swap) operation.

An SDRAM controller works efficiently with large blocks of data and small accesses, such as CAS, should be avoided to enable efficient bounds on bandwidth and response times. CAS is hence more suitable for implementation in the controller of a scratchpad memory (that may or more not be close to the SDRAM controller) and is hence not discussed in this document.

M-4-020 DRAM Configuration:

DRAM configuration is memory mapped within the memory controller.

The interesting configuration parameters in the reconfigurable memory controller (memory patterns, memory map configuration, access granularity, worst-case timing parameters, and arbiter configuration are stored in memory mapped registers that are programmable through the configuration infrastructure.

M-0-062 Memory Performance Counters:

The Memory shall have a cycle counter, which can be read out for performance analysis.

The relevant performance metrics for the memory controller are the actual arrival times, scheduling times, and finishing times of requests. The hardware implementation of the memory controller has prototype support for logging these using a cycle counter, as demonstrated in Section 3.7.3 of this document.

M-5-064 Memory Access Instruction Latency:

The latency of instructions to access main memory shall be latency-bounded.

The memory controller is time-predictable and provides efficient bounds on response times of requests, even during arbiter reconfiguration, as shown in this document.

M-6-041 Bounded Memory Access Time:

Any access to a processor external resource (i.e. memory, NoC) shall execute in bounded time (depending on resource and access time).

See previous requirement.

5 Conclusions

Section 2 described the prototype Manhattan grid network-on-chip (NoC) and tree-shaped network implemented for T-CREST. We discussed the features of these implementations and the possibilities for extending them, so that they may be integrated with the PATMOS CPU and the reconfigurable SDRAM controller in future work packages.

Section 3 presented a general architecture for a reconfigurable SDRAM controller. Three architecture instances with different arbiters and different methods for achieving composability were identified. We discussed the challenges associated with providing interesting reconfiguration operations, such as starting and stopping memory clients and changing the access granularity of the controller, for each of these instances. A detailed discussion was provided on the most reconfiguration-friendly instance, which combined composable memory patterns with TDM arbitration, and a method for time-predictable and composable reconfiguration in presence of persistent memory clients was presented. The proposed approach to reconfiguration was experimentally evaluated and showed that it provides time-predictable and composable behavior during reconfiguration at cost of a slight reduction in guaranteed bandwidth and response times.

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, Williston Hayes Jr., and Kees Goossens. Classification and Analysis of Predictable Memory Patterns. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 367 –376, August 2010.

[6] 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.

[7] 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.

[8] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[9] Bluespec. About the synthesizable modeling company. `http://www.bluespec.com/about/index.htm`, 2013.

[10] 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.

[11] 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.

[12] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks - Past, Present and Future. In *Proceedings of the 10th Workshop on Worst-Case Execution Time Analysis*, July 2010.

[13] 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.

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

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

[16] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A Statically Scheduled Time-Division-Multiplexed Network-on-Chip for Real-Time Systems. In *Proc. IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 152–160. IEEE Computer Society Press, May 2012.

[17] Hardik Shah, Alois Knoll, and Benny Akesson. Bounding SDRAM Interference: Detailed Analysis vs. Latency-Rate Analysis. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2013.

[18] J. Sparsø, E. Kasapaki, and M. Schoeberl. An Area-efficient Network Interface for a TDM-based Network-on-Chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages ??–??, 2013. (Accepted).

[19] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking (ToN)*, 6(5):611–624, 1998.

[20] Jack Whitham and Neil C. Audsley. Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, RTAS '12, pages 3–12, Washington, DC, USA, 2012. IEEE Computer Society.

[21] Xilinx. MicroBlaze Processor Reference Guide. Manual UG081, 2008.

[22] Xilinx. AXI Reference Guide. Manual UG761, 2011.

[23] Xilinx. ML605 Documentation. `http://www.xilinx.com/support/#nav=sd-nav-link-140997&tab=tab-bk`, 2012. [Online; accessed 20-Dec-2012].

[24] Xilinx. Spartan-6 FPGA Memory Interface Solutions User Guide (AXI). Manual UG416, 2012.