T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE FOR EMBEDDED SYSTEMS

Project Number 288008

D 4.5 Integration of Combined Memory Controller

Version 1.0
28 February 2014
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**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the Partners accept no liability for any error or omission in the same.

© 2014 Copyright in this document remains vested in the T-CREST Project Partners.

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 José Neves Av. D. João II, Torre Fernão de Magalhães, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: jose.neves@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 Memory Controller Architecture **3**
- 2.1 SDRAM memory and PHY . . . 3
- 2.2 SDRAM back-end . . . 3
- 2.3 SDRAM front-end . . . 4

3 Memory Tree **6**
- 3.1 Motivation . . . 6
- 3.2 Architecture . . . 7

4 Integration **9**
- 4.1 Memory Controller Integration . . . 9
- 4.2 Patmos Integration . . . 10

5 Prefetcher **12**
- 5.1 Design . . . 12
- 5.2 Evaluation . . . 13

6 Timing Analysis **15**
- 6.1 Memory Controller . . . 15
  - 6.1.1 Worst-Case Situation . . . 15
  - 6.1.2 WCET of a Transaction . . . 16
  - 6.1.3 Pipelining . . . 17
  - 6.1.4 WCET of a Refresh . . . 17
  - 6.1.5 WCET Results . . . 18
- 6.2 Interconnect . . . 18
  - 6.2.1 Single Multiplexor . . . 18
  - 6.2.2 Combined Tree . . . 19
- 6.3 Combined Latency . . . 21

7 Requirements **23**

Document Control

Version	Status	Date
0.1	First draft	10 February 2014
0.2	Second draft	27 February 2014
1.0	Final	28 February 2014

Executive Summary

This document describes Deliverable *D 4.5 Integration of the Combined Memory Controller* of Work Package 4 of the T-CREST project, due 30 months after the project start as described in the Description of Work.

This document presents the integration between the memory-tree subsystem, developed by UoY, and the memory controller provided by TUE as described in D 4.4 in month 24. First, we explain the basic architecture of the memory controller, before giving an overview of the memory tree and a description of the integration of both components. Next, we give an overview of the prefetcher developed for Task 4.4 (*Feedback Control-Based Memory Scheduling*) and finally provide a timing analysis of the system as a whole.

A compiled design has been circulated within the project, noting that it is infeasible to distribute the memory controller publicly for licensing reasons. This compiled design is then intended to be connected to Patmos (from D 2.3) and the T-CREST network-on-chip (from D 3.3) and serve as an evaluation platform on the ML605 FPGA board.

1 Introduction

Current embedded systems are featuring ever-increasing numbers of cores integrated into the same system, all of which require the ability to communicate between each other. In addition to this, the ever-growing storage requirements of embedded tasks typically exceeds the sizes of local memory, requiring the use of external memory, i.e. DRAM. As described in D 4.4, DRAM is not well suited to real-time systems, since the response time depends on the previous access stream, and thus many DRAM controllers cannot provide accurate bounds on the worst-case response time.

This effect is further compounded by the interconnect used to communicate with memory. Bus-based systems do not scale to the requirements of modern systems; the requirement for many-core implies long wire routing costs, leading to slow designs. This is largely the motivation for network-on-chip systems. However, typical network interconnects, such as Manhattan grids and fat trees, introduce added complexity and overheads, which are not required when simply communicating with memory. Standard network-on-chip designs typically also couple the memory and communication requirements of an application and share the interconnect for both memory communication and inter-process communication. This can lead to some applications not being able to communicate efficiently due to others applications tying up network links with memory accesses.

To further add to this complexity, many tasks running within an embedded system have hard deadlines, and thus require a level of predictability within the system. Such predictability is typically achieved by using a predictable arbiter in order to schedule memory requests in a fair and predictable fashion. These arbiters typically split an input stream into a number of buffers, one for each client or priority level. This requires a large multiplexer/demultiplexer arrangement, which simply does not scale as the number of clients increases. his is the motivating factor behind designing a communication network specifically tailored to the requirements of memory access rather than inter-process communication.

The rest of this document is organised as follows. Section 2 contains an overview of the architecture of the dynamically scheduled memory controller from D 4.4, before discussing the Bluetree memory interconnect in Section 3. Section 4 then describes the integration of these two components, while Section 5 describes the prefetcher developed in Task 4.4 that improves the average-case execution time of running applications. Section 6 concludes by showing that the integrated system is time-predictable by providing a worst-case response time analysis of a memory transaction from a processor.

2 Memory Controller Architecture

This section outlines the architecture of the reconfigurable memory controller, previously presented in D 4.2. Figure 1 shows the three main blocks that constitute the memory controller architecture. Working backwards from the SDRAM itself they are the *PHY*, *SDRAM back-end* and the *front-end*. We proceed by discussing the different blocks in more detail.

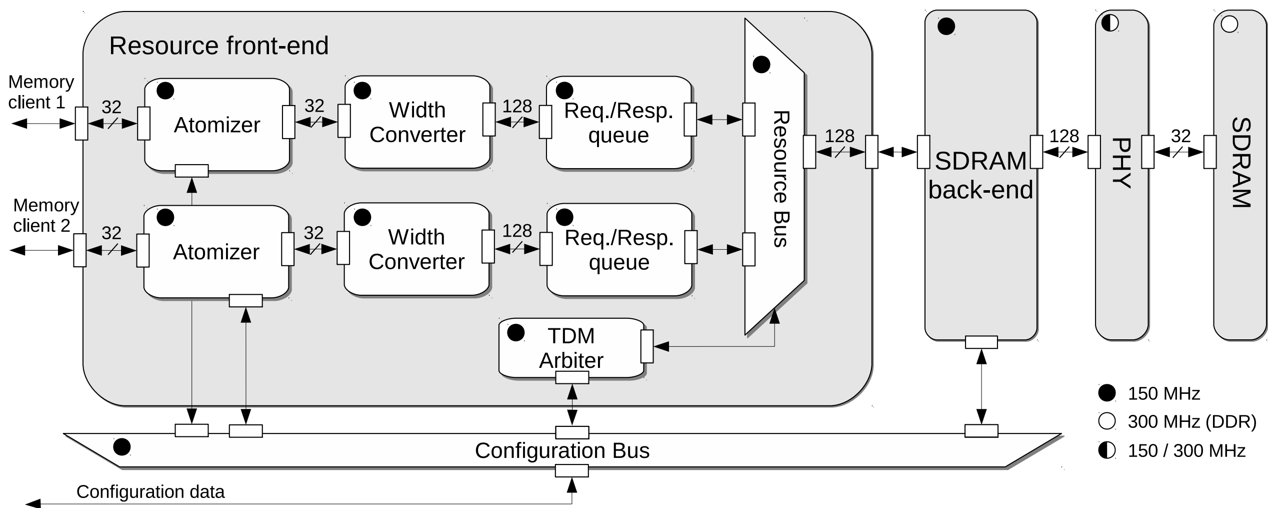


Figure 1: The architecture of the reconfigurable memory controller.

2.1 SDRAM memory and PHY

The memory on the Xilinx ML-605 development board is a 64-bit DDR3-1066 SDRAM, comprising four 1Gb DDR3-1066MHz (-187E) x16 SDRAM chips from Micron [3] on a single-rank SODIMM module. Each of the four chips has a page size of 2 KB. The memory is clocked at 300 MHz, resulting in a clock period tCK of approximately 3.3 ns. According to the timing constraints of the particular memory device [3], the relevant timing constraints are have been calculated and the conservative values are illustrated in Table 1. These are the actual values of the timing constraints used for scheduling memory commands in this deliverable.

The PHY handles the physical I/O connections to the memory and is based on a reference design generated by the Xilinx MIG 3.6 tool. This reference design only uses half of the memory's data pins, effectively resulting in a 32-bit interface. As a result, a single DRAM burst of 8 words corresponds to 32 B of data. Just like the memory, the PHY is clocked at 300 MHz and transfers data on both the rising and falling edges of the clock, achieving a peak bandwidth of 2400 MB/s.

2.2 SDRAM back-end

The SDRAM back-end interfaces with the PHY and is responsible for generating commands to access the memory according to the incoming requests, while making sure that the timing constraints

Table 1: The actual values of timing constraints for 1Gb DDR3-1066MHz (-187E) SDRAM.

TC	*Description*	*Cycles*
tCK	Clock period	1
$tRCD$	Minimum time between *ACT* and *RD* or *WR* commands to the same bank	6
$tRRD$	Minimum time between *ACT* commands to different banks	4
$tRAS$	Minimum time between *ACT* and *PRE* commands to the same bank	16
$tFAW$	Window in which at most four banks may be activated	16
$tCCD$	Minimum time between two *RD* or two *WR* commands	4
tWL	Write latency. Time after a *WR* command until first data is available on the bus	6
tRL	Read latency. Time after a *RD* command until first data is available on the bus	6
$tRTP$	Minimum time between a *RD* and a *PRE* command to the same bank	4
tRP	Precharge period time	6
$tWTR$	Internal *WR* command to *RD* command delay	4
tWR	Write recovery time. Minimum time after the last data has been written to a bank until a precharge may be issued	6
$tRFC$	Refresh period time	44
$tREFI$	Refresh interval	2340

between the commands are satisfied. It translates logical addresses to a physical bank, row and column in the memory, and it also refreshes the memory periodically every $tREFI$ cycles. This part runs at 150 MHz and hence has to supply the PHY with two commands and four 32-bit data words per clock cycle. The latter is achieved by using a wider data width of 128 bits in the back-end.

The back-end architecture is based on the dynamically scheduled memory controller delivered in D 4.4. The implementation is done in VHDL and includes 5 pipeline stages, as shown in Figure 2. After a transaction arrives at the interface of the back-end, the relevant information is obtained in Stage 1, including the transaction type (read or write), logical address and the required number of bursts. Stage 2 plays a role in splitting the transaction into the required number of bursts, and getting the physical address of each burst by address decoding. Thereafter, the *command generator* in Stage 3 produces the required *ACT*, *RD* or *WR* and *PRE* commands for each burst. Note that all transactions are assumed to be aligned with a row in the bank. The generated commands are then inserted into the *command FIFO*. In Stage 4, the *timing selector* is responsible for checking the associated timing constraints for scheduling the command at the head of the command FIFO. If the timing constraints are satisfied, the command is issued by the *command scheduler* in Stage 5. The action in each stage consumes one clock cycle, except the command generation in Stage 3, which takes one cycle per command and hence consumes several cycles to generate the commands required for a burst.

2.3 SDRAM front-end

The primary function of the resource front-end is enabling sharing of the SDRAM. The *atomizer* splits incoming memory requests into fixed size chunks called *atoms*. This allows clients to be preempted at the granularity of atoms, independently of their actual request behaviour. The size of an atom corresponds to the granularity at which the back-end handles requests, referred to as the *access granularity* of the memory controller. The access granularity typically ranges from 16 B up to 1 KB,

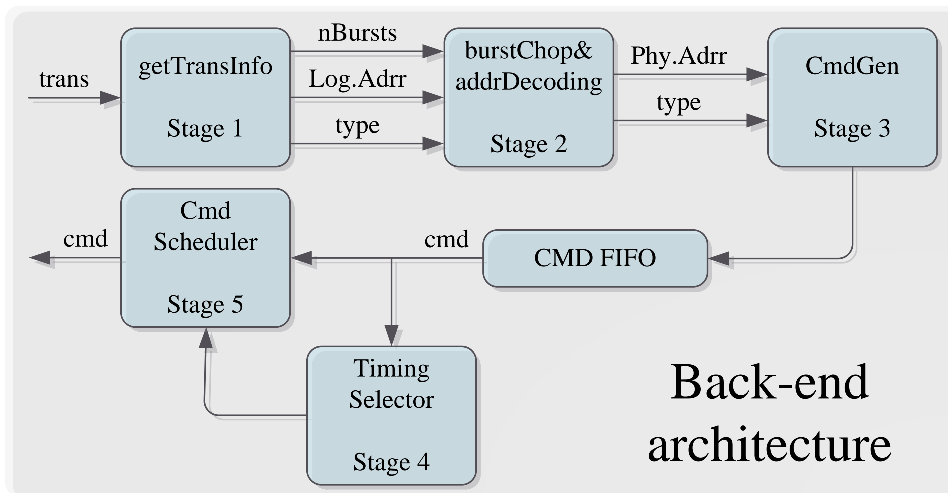


Figure 2: Architecture of the dynamically scheduled back-end.

depending on the memory and the controller configuration. The *width converter* accepts messages using small data words from the memory clients (generally 32 bits wide), and converts them to the 128-bit words used in the back-end. In essence, this is a common serial-to-parallel converter. The *request/response buffer* holds incoming atom-sized requests until either all data is buffered (for write requests), and enough space is available for the response (for read requests). This is required because data has to be provided to and accepted from the back-end without blocking according to the JEDEC DRAM specification [1]. A request is only eligible for scheduling once this condition is met. The *reconfigurable TDM arbiter* schedules one of the eligible requests from the request buffers to be processed by the back-end, and optionally inserts an idle slot if no request is available. The memory is shared at a fine granularity, such that each slot corresponds to one read or write atom. The *configuration bus* allows various memory-mapped registers to be programmed by a configuration host, typically a processor. All blocks in the front-end are clocked at 150 MHz.

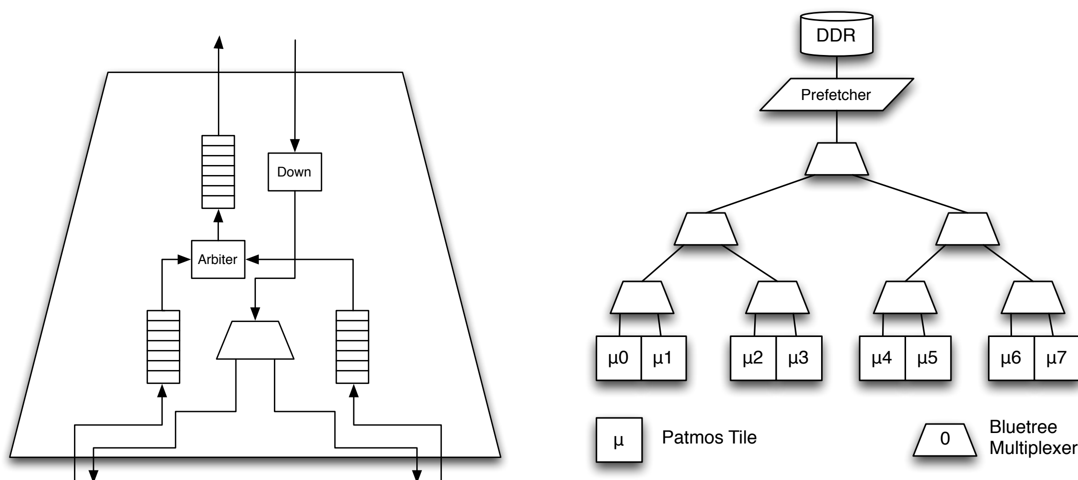
3 Memory Tree

3.1 Motivation

The memory tree is a memory interconnect motivated by the growing bandwidth requirements of embedded systems and the need to be able to decouple the memory requirements of an application and its communication requirements; a network mapping which meets all communication deadlines for a given set of tasks may not be able to allow all high-bandwidth memory requestors to meet their requirements, and vice versa. In addition, if a requestor on the network requires a high amount of memory bandwidth, this can then block other clients from attaining their communication requirements in both TDM-based systems (since the memory requestor will require a large number of scheduling slots) and in standard X-Y routed best-effort systems (since the memory requestor will occupy shared links for a large amount of time).

In addition, standard monolithic arbitration schemes cannot scale to the number of cores found in modern systems. In order to support a large number of clients, demultiplexing the incoming requests into a number of virtual channel buffers is required before these are then multiplexed back onto the path to memory. As the number of cores increases, the size of these multiplexors and the time taken to cross them increases, and hence the maximum clock frequency of the interconnect decreases.

For these reasons, a second network was developed in order to handle memory traffic, leaving the TDM-based mesh NoC to be used solely for inter-process communication. This tree is tuned for the requirements of memory requestors in that it does not allow for communication between cores, only to and from memory, and also features wide buses and packets for the fast transfer of large amounts of data. In addition, the multiplexor-based design allows for memory arbitration to be distributed into the routers themselves, rather than having a single global arbiter attached to the memory tile. This removes the need for a large multiplexer/demultiplexer arrangement.



(a) Internals of a memory tree multiplexor

(b) Example of a composed memory tree design

Figure 3

3.2 Architecture

The memory tree is composed as a set of 2-into-1 full-duplex multiplexors. On the path upwards (to memory), there is a static priority ordering favouring the left hand side of the multiplexor. On the path downwards, a packet must not be blocked. Since there is only one server at the top of the tree, which can only create a single packet per cycle, there is no arbitration required on the path down from memory, and thus it is possible to have non-blocking communication on the downwards path.

Of course, this static priority can in fact lead to a single client dominating the whole tree and thus locking out all other transactions. Assume the case where the leftmost client issues a request to memory on every cycle. This will cause this client to take priority on every cycle and thus lock out all other clients. While this is not a feasible scenario for a processor, this can still happen in real systems. In a sufficiently large system, there may be a sufficient number of fetches issued such that a *sub-tree* can dominate one side of a multiplexor located at a high position in the tree.

This is fixed with a simple arbitration scheme. This encodes a "block counter" in the router, which stores the number of high priority packets which have been transmitted while a low-priority packet has been blocked. When this counter is equal to a fixed value, then a single low priority packet is permitted to be transmitted. This arbitration scheme is described and analysed in more detail in Section 6.2.

A diagram of this architecture can be found in Figure 3a. To compensate for blocking, both input ports and the output ports on the upwards path are buffered with FIFOs. Since the down path is non-blocking, a FIFO is not required, instead, the downwards packet is simply stored in a register (i.e. "down" in Figure 3a). This is then demultiplexed back onto both downwards outputs based on the *CPU ID* field of the packet, or can be broadcast to both. Since there is no buffering on the downwards path, a memory tree client must always be able to receive a packet on every cycle, otherwise data may be lost. There is no such restriction on the upwards path; since a packet may be blocked, the upwards path is fully buffered and hence is able to stall. These multiplexors are then composed into a tree form, an example of which can be seen in Figure 3b. Due to the routing algorithm used, these trees do not need to be balanced.

4 bit	128 bit	16 bit	28 bit	8 bit	8 bit	4 bit	4 bit
Type	Data	BEN	Address	Task ID	CPU ID	Priority	Size

(a) Memory tree client packet format

3 bit	128 bit	28 bit	8 bit	8 bit
Type	Data	Address	Task ID	CPU ID

(b) Memory tree server packet format

Figure 4

These multiplexors route 198-bit packets up the tree to the server, and 172-bit packets back down to clients. The layout of these packets can be found in Figures 4a and 4b. These packet formats were

chosen in order to keep the multiplexors simple; since everything is contained in a single packet, there is no need for logic to deal with variable sized packets at the multiplexor level. The fields within the packets are then broken down as follows:

Type: The type of the request. For client messages, this is used to distinguish between standard requests and prefetches. For server messages, this is used to distinguish between returned read data, write acknowledges and different prefetch types.

Data: The data read from, or to be written to memory.

BEN: Byte-enable. If this field is zero, it implies that the packet is a read, if non-zero, then the bytes in *data* for which the bit in *BEN* is 1 should be written to memory.

Address: The address to read to or write from, aligned on a 16-byte boundary.

Task ID: Reserved for future expansion. Intended to specify client specific information.

CPU ID: The client ID. This is filled in automatically by the multiplexors. Each multiplexor will shift this field, then add the direction that the request came from (e.g. 0 for left and 1 for right). When delivering a response, the reverse procedure is carried out in order to deliver the response to the correct client.

Priority: Reserved for future expansion. Intended to specify the priority for multiplexors that honour this field.

Size: Used for reads. Allows for multiple 16-byte words to be read from sequential memory locations. If this field is $n > 0$, then the memory controller will deliver n 16-byte responses from addresses $addr, addr + 1, ..., addr + n$.

4 Integration

4.1 Memory Controller Integration

The integration of the memory tree with the dynamic memory controller can be done in one of two ways; either it can be implemented as a client onto one of the memory controller's ports (i.e. memory client n in Figure 1), or by interfacing directly onto the back-end of the memory controller (i.e. replacing the whole "Resource front-end" in Figure 1). Both methods will result in losing the full capabilities of the arbiter at the memory-side; demultiplexing the tree onto the standard client ports would ultimately negate the gains given by the memory tree as a whole. The former is a simpler solution since the memory tree can simply act as a single client, yet wastes logic resources since the arbiter is now effectively not doing anything. The latter more difficult and requires emulating the communication between the front and back end of the memory controller. The latter approach was eventually chosen though in order to remove this redundant logic, providing a tighter, faster and smaller system.

The predictable memory controller uses the DTL bus format internally, hence a bridge from the memory tree onto DTL is required. In addition, since the memory back-end expects atomised packets, an atomiser is also required in order to resize the input request into correctly sized packets. In the case of this memory controller, atoms with a minimum size of 512 bits are expected.

The width converter in Figure 1 builds 128-bit requests from multiple 32-bit packets, and is also responsible for splitting a 128-bit response into multiple 32-bit response packets. Since this is already the size of the data width in the memory tree, the DTL bridge as a whole can be much simplified.

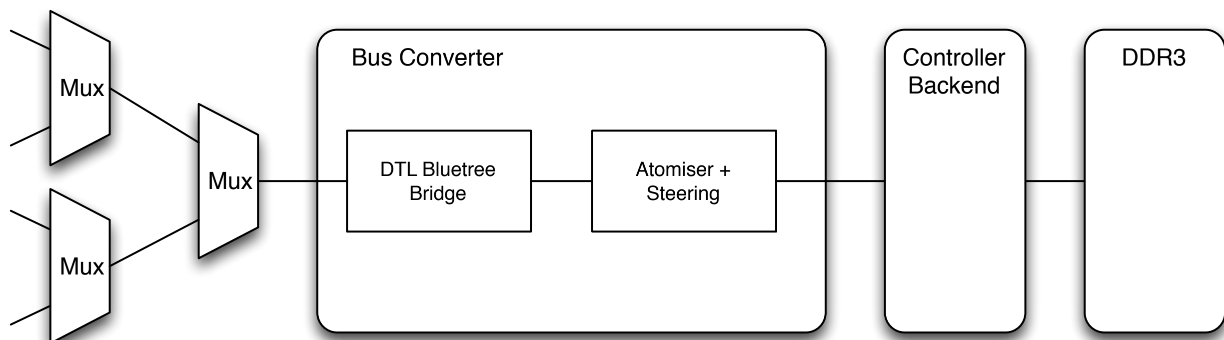


Figure 5: Integration between the memory tree and the predictable memory controller

The architecture of the DTL bridge is then shown in Figure 5. The DTL to memory tree bridge translates memory tree packets into a form more suitable for DTL, and vice versa for the reverse path. The atomiser then breaks these packets into atoms, as in the original design. In addition though, this must also do some byte-steering. This is because the predictable back-end expects requests to be aligned on an atom-boundary, in this case 64-bytes, and to read or write a full atom. Hence, extra logic is required in order to mask unnecessary data (both for reads and writes) and to ensure the data to be written appears at the correct offsets.

Any excess data from an atom that is not requested is simply discarded. While this appears wasteful, this discarded data does not carry a great performance penalty. Since DDR3 memory has a fixed burst length of eight, and the controller only utilises 32-bits of the available 64-bit memory path, eight consecutive 32-bit words are always provided by the SDRAM module, hence no extra data is requested then discarded. Since the memory tree supports variable sized reads, the overall system performance can be improved by increasing the amount of data read in a single transaction, hence utilising the data which previously would have been discarded.

Finally, many bus standards require a slave to acknowledge that they have received the transaction, even if no data has been generated (for example, on a memory write). This acknowledge can be represented in the memory tree by replying with a write acknowledge packet. When issuing a memory write to the memory controller back-end, the DTL bridge will respond to the client with this acknowledge packet when the last word of data has been issued to the memory back-end. Since memory transactions will not be reordered by the back-end, the written data must have been written to memory when this acknowledge is generated and thus the system remains in a coherent state.

4.2 Patmos Integration

While Section 4.1 covers integrating memory into the T-CREST system, the Patmos processors, as described in D 2.3, must also be combined with the memory subsystem in order to create a full evaluation system.

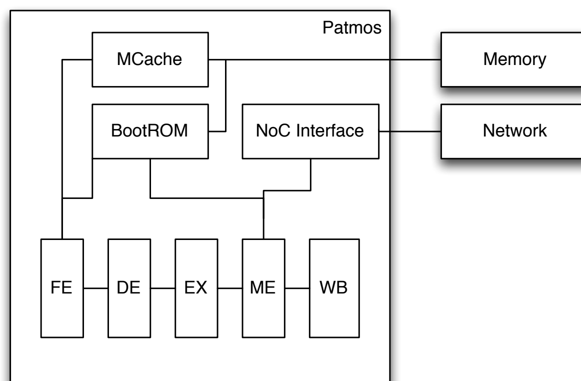


Figure 6: Internal diagram of Patmos buses.

Patmos itself exposes three top-level ports for communication with the outside world, namely an *OcpIO* port intended for communication with the network arbiter's control port, an *OcpCore* port intended to communicate actual data to and from the network, and finally an *OcpBurst* port for communicating to and from memory. This is then connected internally to the method cache and to the memory stage in the pipeline via its internal ROM. This can be seen graphically in Figure 6.

This *OcpBurst* memory port is a 32-bit Ocp port, with the burst length tied off at 4 words, implying that Patmos' native read/write data width is 128 bits - identical to the memory tree. Since the interface is 32 bits wide, these data words are transferred on subsequent clock cycles to and from the processor.

A similar method to integrating the memory controller is then employed; an OcpBurst to memory tree adapter has been created, which is responsible for constructing a memory tree packet from four OcpBurst bursts, and vice versa for the reply path.

5 Prefetcher

5.1 Design

In order to attempt to alleviate some of the latency of a memory request, i.e. bus arbitration time, tree traversal time and DRAM latency, a prefetcher has been developed for this system. This is intended to be added to the system at the root of the tree and speculatively issue memory accesses at run time based upon the current reference stream. In the perfect case, this would then deliver required words to the target cache of the processor *just* as the word is required for processing. If this deadline is missed slightly, then simply issuing the request is sufficient to remove some of the latency of a full memory request, hence improving execution time still.

By sitting at the root of the tree, the prefetcher can ascertain information about the reference stream as a whole, and additionally ascertain information about the current total memory load of the system. The prefetcher implements a stream prefetcher [2] and makes the assumption that if a processor requires memory addresses n, $n + 1$, $n + 2$ etc in sequence, it will likely also require address $n + 3$ and $n + 4$ in the near future, and thus those should be pre-fetched.

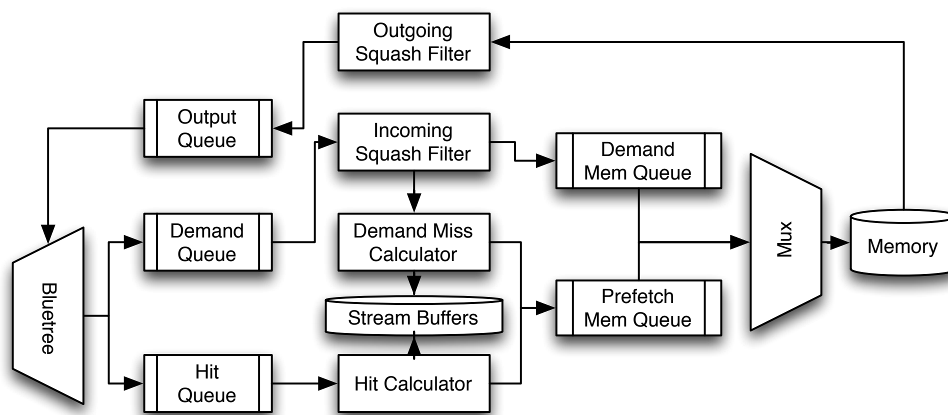


Figure 7: Internal diagram of the prefetcher

The prefetcher itself contains a number of components, which are shown in Figure 7 and described below:

Demand Queue: Stores incoming demand memory requests, i.e. cache misses and stores.

Hit Queue: Stores incoming prefetch hit notifications. These are generated when a prefetched line is used by the target processor.

Incoming Squash Filter: This is used in conjunction with the "Outgoing squash filter" component. These are used in order to coalesce memory accesses with prefetches. If a line is currently being prefetched and a demand miss arrives for it, this filter adds a new record to a "squash table" (not pictured for brevity). The output squash filter then examines all returning prefetches, and

if one matches an entry in the squash table, it is returned to the processor as a read rather than a prefetch. This then reduces the latency of a read request and also eliminates a redundant request.

Demand Miss Calculator: This is responsible for correlating streams in the stream buffers. This looks up the *previous* address in the stream buffers. If an entry is found with that address, it is updated and a prefetch sent for the *next* address. If no entry is found, one is added with the *current* address.

Hit Calculator: This is identical to the "Demand Miss Calculator", but searches for the *current* address. If a record is found, it is updated and a prefetch dispatched for the next line.

Demand Mem Queue: This then buffers outgoing memory requests separately to the outstanding prefetch requests. The "mux" component then multiplexes memory requests onto the memory controller with a priority ordering favouring demand misses. This is such that a prefetch cannot be dispatched before a required memory line and thus prevent damage to system responsiveness.

Prefetch Mem Queue: See above.

Outgoing Squash Filter: See "Incoming Squash Filter".

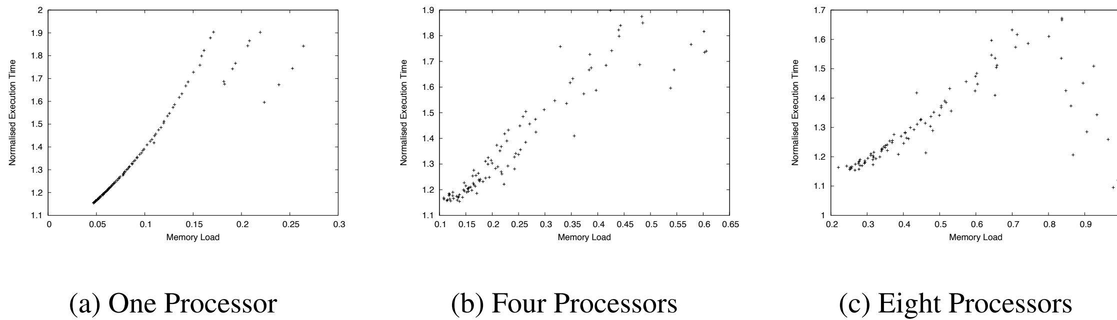
Output Queue: Used to buffer words returned from memory, before sending them back down the tree.

Since this prefetcher inserts data directly into the target processor's cache, and requires feedback when a prefetch is used by the processor, modifications to the processor cache are required. This cache must be able to accept data being pushed into it which has not been requested (hence requires a dual-ported cache store) and also must track which lines have been pushed in. Finally, it should send a memory request of type *prefetch hit* when a hit occurs on these lines, which is then caught by the prefetcher.

5.2 Evaluation

This prefetcher was implemented onto an ML-605 evaluation board with a standard memory controller for testing. It was connected to eight Microblaze processors, each of which was running a basic traffic generator program. This program simply reads a cache line from memory, then holds off for a set number of cycles in order to simulate a delay. It then repeats this process over a number of iterations.

The system was evaluated with the prefetcher enabled and disabled, then across hold-off delays from 3 to 300 cycles. In addition, logic was added to the memory bridge to ascertain the percentage of the total system time that the memory controller was occupied. Logically, as the delay between accesses decreases, the total system load should increase. Results were then plotted on the graphs as seen in Figure 8.



(a) One Processor (b) Four Processors (c) Eight Processors

Figure 8: Plots of speedup vs. memory load for the prefetcher enabled.

The graph for one processor (Figure 8a) shows a sloping upwards curve, showing the system performance improving as memory load increases (and thus the delay decreases). In actuality, this is a constant speedup, since the memory hold-off delay does not change with the prefetcher enabled or disabled, hence the hold-off delay constitutes most of the execution time. The small “tails” caused towards the right of the graph are actually caused by bus delay. Since the prefetcher can only coalesce prefetches which have *not* yet re-entered the prefetcher, it cannot coalesce those which are on the path back from the prefetcher to the processor. These “tails” are then caused when a prefetch is in-flight back to the processor when a demand miss arrives for the same line.

The graph for four processors (Figure 8b) shows a similar trend. This is an artefact from the transmission delay to and from the memory. If this transmission delay is great enough, it is possible for tasks to become interleaved. As an example, while Task 0 has a memory request travelling down the tree, Task 1 may have a memory request being serviced while Task 2’s request is also still travelling up the tree. This then leads to a graph of similar shape to that for one processor.

Finally, with eight processors, the effect of memory over-load starts to become apparent. As the load on memory increases, prefetches start to get dispatched too late (due to the priority ordering). Since a prefetch cannot be cancelled, this causes the prefetch to take up memory bandwidth with data which has already been fetched, effectively slowing down overall system performance.

6 Timing Analysis

This section shows that the integrated system is time-predictable by presenting a response time analysis for a memory request from the processor. The analysis derives the total worst-case response time (*WCRT*) of a single outstanding memory request from a memory client, corresponding to the case where a requestor is stalling for every memory access. Since the Patmos processor stalls on a memory access, this assumption is in line with the platform being developed.

6.1 Memory Controller

First, we present a timing analysis of the dynamically scheduled memory controller back-end connected to the memory tree. The goal is to determine the maximum time a request can spend executing in the memory controller, i.e. the local *WCRT* of the request. This result is later added to the *WCRT* of the memory tree in Section 6.3 to form the total *WCRT* of the request. The *WCRT* of a read request is defined as time from the arrival of the read command until the first word of response data is available on the memory controller interface. Memory controller interface in this context considers the interface of the back-end, since the front-end is not used. For a write request, the *WCRT* is defined as the time from the arrival of the write command and first data word on the memory controller interface (assumed to happen in parallel) until the last write command has been issued to the memory.

As previously explained in Section 2, the memory controller contains multiple clock domains, making the term clock cycle ambiguous. Unless explicitly mentioned otherwise, the presented analysis is based on the clock frequency of the memory device, i.e. 300 MHz in this section.

The timing analysis of the memory controller is structured as follows. First, we will derive the worst-case situation for a transaction arriving in the back-end. Then, we proceed by determining the worst-case execution time *WCET* of the request, corresponding to the maximum time required to schedule all of its DRAM commands. We then continue by considering extra delay introduce by pipeline stages in the architecture of the back-end, PHY, and memory device, since they affect the *WCRT* of the request. Lastly, we discuss the impact of refresh on the execution time of the application and conclude with some results.

6.1.1 Worst-Case Situation

An arbitrary transaction T_i ($\forall i > 0$) arrives at the back-end of the memory controller with the arrival time that is given by Definition 1. The arrival time is defined as the time at which T_i arrives at the interface of the back-end. Moreover, Definition 2 defines the finishing time of T_i as the time when its last *RD* or *WR* command $RW_{last}^{T_i}$ has been issued. The starting time of T_i is defined as the earliest time at which the scheduler tries to issue its commands (Definition 3). This is either one cycle after the finishing time of the previous transaction T_{i-1} or the arrival time, whichever is larger. Lastly, the difference between the finishing time and the starting time is referred to as the execution time of the transaction, defined in Definition 4.

Definition 1 (Arrival time)**.** *$t_a(T_i)$ is defined as the time at which T_i arrives at the interface of the back-end.*

Definition 2 (Finishing time)**.** $t_f(T_i) = t(RW^{T_i}_{last})$

Definition 3 (Starting time)**.** $t_s(T_i) = max\{t_f(T_{i-1}) + 1, t_a(T_i)\}$

Definition 4 (Execution time)**.** $t_{ET}(T_i) = t_f(T_i) - t_s(T_i) + 1$

In order to obtain the *WCET* of T_i, its starting time should be as early as possible, while the finishing time must be maximised according to Definition 4. As Definition 3 indicates the execution of T_i starts either immediately after the finishing of T_{i-1} or its arrival, the former guarantees a larger finishing time of T_i, since the command scheduler has to wait longer time to issue a command of T_i to ensure that the timing constraints are satisfied. In contrast, if the starting time of T_i is equal to its arrival time, the command scheduler takes less time to issue a command of T_i. The reason is that the timing counters have already been recounting since the last command of T_{i-1} has been issued before the arrival of T_i. Moreover, the three pipeline stages in Figure 2 that T_i experiences from its arrival to inserting its first command into the command FIFO consume only 3 cycles. However, the minimum time that the command scheduler has to wait until issuing the first command of T_i immediately after T_{i-1} finishes is *tCCD*, which is typically 4 cycles according to the Micron specification [3]. As a result, a larger finishing time of T_i cannot be obtained if its starting time is equal to its arrival time. Consequently, the worst-case situation for T_i is that the command scheduler immediately starts trying to schedule the commands of T_i when T_{i-1} finishes.

6.1.2 WCET of a Transaction

A transaction T_i is executed by scheduling the required commands to read or write a number of bursts (*nBursts*), determined by the size of the request. In the worst-case situation, the first burst requires issuing a *PRE* command to close the currently opened row, and then an *ACT* command follows to open the target row. Finally, each burst uses a *RD* or *WR* command to trigger the data transmission from/to the memory. In addition, the previous transaction T_{i-1} is a write rather than a read, because the timing constraint for issuing a *PRE* of T_i immediately after the previously scheduled *WR* command is larger than for a *RD* command. This timing constraint is $tRTP$ that is much smaller than $tWTP$, as given by Equation (1). The corresponding timing constraints are given by the Micron specification [3], and *BL* is the burst length (8 words). The meaning of the timing constraints was previously summarized in Table 1.

$$tWTP = tWL + BL/2 + tWR \quad (1)$$

Since the worst-case situation for T_i is that the command scheduler starts trying to schedule the first *PRE* command of T_i immediately when the previous write transaction T_{i-1} finishes, T_i is executed by waiting $tWTP$ cycles to schedule the *PRE* command, followed by scheduling the *ACT* command after the precharge period time (tRP) and then *nBursts* number of *RD* or *WR* commands being scheduled. According to Definition 4, the worst-case execution time of T_i is obtained and is shown by Equation (2).

$$\hat{t}_{ET}(T_i) = tWTP + tRP + tRCD + (nBursts - 1) \times tCCD \quad (2)$$

6.1.3 Pipelining

The execution time of a transaction considers the time required to schedule all its memory commands, but it does not account for pipeline delays. While it is possible to include this in the execution time, it is not desirable if the execution time is used to compute the maximum interference from other memory clients by multiplying it with the number of interfering requests. This is because such a computation would fail to capture that previous and later requests may be traversing the pipeline stages of the back-end, PHY, and memory device, while the commands of the current request are being scheduled. The pipelining is hence represented by a constant offset value, Δ, that is added to the *WCRT* of a transaction. We proceed by explaining how to derive the offset values for read and write transactions, respectively.

There are 5 pipeline stages on the request path, which cover the stages from arrival at the interface of the back-end until the first SDRAM commands are accepted by the PHY. If a read command is presented to the PHY in Stage 5 then the first data comes back 13 cycles later, due to read latency in the memory and internal stages in the Xilinx PHY and memory device where nothing visibly happens in the back-end. This number has been derived by measurements. Note that this measurement contains a variable component that is determined by the PHY during initialisation. This means it could vary depending on the memory module, FPGA board, and environment, although this is currently not considered. Lastly, there is a single pipeline stage on the response path of the memory controller.

Stages 1-5 are on the request path in the back-end, while there is one pipeline stage on the response path. Write requests only experience the stages on the request path, while reads go through all pipeline stages on both request and response paths. However, these stages are clocked at 150 MHz and hence take 2 clock cycles at 300 MHz. For reads, we must also consider the extra 13 cycles from scheduling the last read until the first data comes back. In total, the offset is computed according to $\Delta_{wr} = 10$ and $\Delta_{rd} = 25$ for writes and reads, respectively.

6.1.4 WCET of a Refresh

To issue a *REF* command, all the opened banks have to be precharged by scheduling a precharge all (*PREA*) command. After the precharge period (tRP), all the banks are precharged and the *REF* command can be issued. The scheduling of a *PRE* command has to satisfy the timing constraints of $tRAS$ and$tRTP$ or $tWTP$. $tRAS$ is a timing constraint between an *ACT* command and a *PRE* command to the same bank, while $tRTP$ or $tWTP$ is the timing constraint between the last *RD* command or *WR* command and the *PRE* command to the same bank. As $tRCD$ is the timing constraint between an *ACT* command and the first *RD* or *WR* command, we can observe from the JEDEC DDR3 standard [1] that $tRAS \leq tRCD + tWTP$ and $tRTP < tWTP$, where $tWTP$ is given by Equation (1). These observations are also true for our 1Gb DDR3-1066MHz (-187E) x16 SDRAM, where the timing constraints are given by the Micron specification [3]. As a result, the longest time for scheduling a *PRE* command occurs if it follows a *WR* command rather than a *RD* command. Therefore, the scheduling of a *PRE* command consumes more time to satisfy the timing constraints

if the previously issued command is a *WR* rather than a *RD*. The worst-case execution time of a *REF* command is hence shown in Equation (3).

$$\hat{t}_{ET}^{REF} = tWTP + tRP + tRFC \tag{3}$$

These values allow us to compute the *refresh overhead*, o^{ref}, which is a multiplier for the *WCET* of the application that accounts for delays caused by refreshing the memory. This delay is considered in the analysis of the application instead of the analysis of the memory controller, since refresh is only required every 7.8 μs for DRAMs at normal operating temperatures. Accounting for this delay for every request hence results in very pessimistic analysis. The refresh overhead is computed according to Equation (4).

$$o^{ref} = \frac{tREFI}{tREFI - \hat{t}_{ET}^{REF}} \tag{4}$$

6.1.5 WCET Results

According to the above analysis, a new transaction suffers a *WCET* only if the command scheduler immediately starts trying to issue its commands after the previous write transaction finishes. The worst-case execution time of a transaction is given by Equation (2). It can be seen from these equations that the *WCET* of either a transaction or a refresh is only determined by the actual values of the timing constraints and the number of required bursts. The *WCET* of transactions with variable sizes are obtained by using the timing constraints presented in Table 1. Consequently, Table 2 provides the *WCET* of transactions with the size of 32 B, 64 B and 128 B, which are supported by the implemented back-end architecture with dynamic command scheduling. The complete response time of a request in the memory controller is obtained by adding the *WCET* in Table 2 and add the offset $\Delta_{wr} = 10$ and $\Delta_{rd} = 25$ for writes and reads, respectively.

In addition, according to Equation (3), the worst-case execution time $\hat{t}_{ET}^{REF}$ consumed by issuing a *REF* command is 66 cycles. This is used to compute the refresh overhead, o^{ref}, in Equation (4), which is approximately 3% with the memory timings used in this deliverable.

Table 2: WCET (cycles @ 300 MHz) of transactions with different sizes.

Size (bytes)	*32*	*64*	*128*
WCET (cycles)	28	32	40

6.2 Interconnect

6.2.1 Single Multiplexor

The cost of crossing the memory interconnect is calculated as the sum of the cost of crossing the upwards path towards memory, the downwards path towards the processors and finally the blocking

time while a low-priority packet waits for arbitration. The equation for the worst-case time to cross the a multiplexor is then defined in Equation (5).

$$t_{mux} = t_{up} + t_{down} + t_{block} \tag{5}$$

First, we express the time taken for a packet to traverse up and down the tree, ignoring arbitration for the moment. This is then a constant value which represents the time taken to be enqueued and dequeued from the FIFOs and registers within a multiplexor. Given that the time taken to be enqueued into a FIFO or written into a register (and be able to be read again) is a single cycle, and referring to Figure 3b, the time for these constants can then be derived as follows:

$$t_{up} = 2 \times t_{FIFO} = 2 \tag{6}$$

$$t_{down} = t_{reg} = 1 \tag{7}$$

These times are the latency from a packet being accepted by the multiplexor to being available on the respective output. For t_{up}, cycle n accepts an incoming packet and enqueues it into one of the lower FIFOs. Cycle $n + 1$ then takes this packet from the top of the FIFO, passes it through the arbiter and enqueues it into the upper FIFO. On cycle $n + 2$, this data is available from the top of the multiplexor, which can be accepted and enqueued into the next multiplexor up (forming cycle n for the next multiplexor).

For the t_{down} value, there is only a single storage element, and non-blocking transfers are assumed. On cycle n, data is stored in the *down* register, and is available on cycle $n + 1$, ready to be stored into the *down* register of the next multiplexor, again, forming cycle n for the next multiplexor.

The value for t_{block} is calculated depending on the static priority of the current requestor into the multiplexor (i.e. if it is high or low priority) and the current maximum blocking time $m > 1$, and is described in Equation (8). This blocking time expresses the time at which, assuming a fully loaded multiplexor, a low priority packet can take precedence over a high priority one. It then follows that a low priority packet can be blocked by at most $m - 1$ high priority packets in this system.

$$t_{block} = \begin{cases} 1 & \text{High Priority Input} \\ m - 1 & \text{Low Priority Input} \end{cases} \tag{8}$$

Since a high priority packet can only be blocked a single time by a low priority packet, it is possible to degrade this system into standard round-robin by setting $m = 2$, implying that a low priority task can only be blocked by a single high priority task. Doing so also removes the complexity of choosing valid values for the m term and eliminates the mapping problem, since memory access delays are (for balanced trees) uniform in a round-robin system.

6.2.2 Combined Tree

Of course, the current analysis only concerns itself with a single multiplexor. Systems with more than two processors are, of course, going to require multiple levels of multiplexors. In order to obtain

this estimate, we must now define a worst-case time for a multiplexor at *depth* d, where a depth of one is the root of the tree and a depth of $log_2(n)$ are the leaf multiplexors for a fully balanced tree with n clients.

Since the root of the tree cannot be blocked by an upstream multiplexor, the analysis of t^1_{mux} is identical to the analysis above. A packet which has arrived at a multiplexor of level 2 then has the following blocking time in Equation (9). The priority levels refer to the current "priority path". High-high implies a client which is of high priority at depth 2 and depth 1, whereas low-high implies a client which is of low priority at depth 2 and high priority at depth 1, and so on.

$$t^2_{block} = \begin{cases} 3+1 & \text{High-High Priority} \\ 2m+m-2 & \text{High-Low Priority} \\ 2m-1+1 & \text{Low-High Priority} \\ m^2+m-2 & \text{Low-Low Priority} \end{cases} \tag{9}$$

The derivation of this is purely logical. Assume the case for high-high and a packet arrives just after the multiplexor at level 2 has transmitted $m-1$ high-priority packets and we are a high priority requestor. In cycle 1, the multiplexor at level 2 will be blocked for a cycle by the multiplexor at level 1, since the quota has been exhausted. In cycle 2, the multiplexor at level 2 will be given service, but a low priority packet will instead take precedence. In cycle 3, the level 2 multiplexor can again be blocked by the level 1 multiplexor for a sufficiently small m (i.e. $m=2$), hence the blocking at level 2 is three cycles. After crossing this multiplexor, we may then again be blocked at the level 1 multiplexor, hence bringing the total blocking term to four cycles.

Assume the high-low case also. Because the level 2 multiplexor is of low priority, it will only be able to relay a packet once every m cycles. Assuming again that the multiplexor at level 2 has relayed $m-1$ packets already, we can be blocked for $2m-1$ cycles; one full cycle of m is due to the level 2 mux relaying a low priority packet, then $m-1$ cycles must be waited in order to gain service again. Finally, the packet may be blocked again at the level 1 mux by $m-1$ cycles.

It can therefore be seen that the blocking factor at a multiplexor depends on the blocking at higher order multiplexors, and is the sum of the blocking factors at each level. We can therefore express t^i_{block} as in Equation (10), where the generic blocking time of a multiplexor at level i, $t^{i\prime}_{block}$ is expressed in Equation (11)

$$t^i_{block} = \sum_{n=1}^{i} (t^{n\prime}_{block} - 1) \tag{10}$$

$$t^{i\prime}_{block} = \begin{cases} 2 & i=1 \land \text{High Priority} \\ m & i=1 \land \text{Low Priority} \\ 2 \times t^{i-1\prime}_{block} & i \neq 1 \land \text{High Priority} \\ m \times t^{i-1\prime}_{block} & i \neq 1 \land \text{Low Priority} \end{cases} \tag{11}$$

Since the crossing times are constant, the total round-trip time for a multiplexor at depth i is then derived as in Equation (12), with t^i_{block} expressed in terms of memory transactions.

$$t^i_{mux} = (t_{up} \times i) + (t_{down} \times i) + t^i_{block} \tag{12}$$

6.3 Combined Latency

This round-trip time is then simply encoding the round-trip time for a single request in the memory interconnect, assuming that a request is immediately enqueued back into the tree as soon as it has been received by the controller at the root of the tree. This creates a factor t_{mem}, which is zero in the derivation of t^i_{mux}. This also assumes that the blocking time at a multiplexor is in terms of cycles; since the memory controller operates on the granularity of memory atoms, then each request will be blocked for an amount of time expressed in terms of atoms.

Given a processor connected to a multiplexor at level i, and given that the amount of data in an atom A that can be requested is a constant $nBursts$, then it follows that the worst case response time for a single memory request can be computed as in Equation (13).

$$t^i_{mux} = (t_{up} \times i) + (t_{down} \times i) + (t^i_{block} \times \hat{t}_{ET}(A)) + \hat{t}_{ET}(A) + \Delta_{rd/wr} \tag{13}$$

Finally, a scaling factor must be applied to the time taking to cross the tree multiplexors, since the tree operates in a different clock domain to the rest of the system. The tree is clocked at 100MHz, whereas the analysis for the memory controller assumes a 300MHz clock and thus is expressed in clock cycles at 300MHz. This is represented by simply scaling t_{up} and t_{down} by the constant factor f_{scale}, hence the response time taking into account the clock difference is computed as in Equation (14).

$$t^i_{mux} = (t_{up} \times i \times f_{scale}) + (t_{down} \times i \times f_{scale}) + (t^i_{block} \times \hat{t}_{ET}(A)) + \hat{t}_{ET}(A) + \Delta_{rd/wr} \tag{14}$$

We can now derive examples for the worst case execution time of a request. Given that the back-end has a minimum request size of 32 bytes, the WCET of a memory request is presented in Table 2 as 28 cycles with a maximum offset of $\Delta_{rd} = 25$. We can then derive the worst case round trip times for a memory request using this result and Equations (13) and (11). These figures assume a fully balanced tree, assume a scale factor $f_{scale} = 3$ (to account for the tree running at 100MHz while the rest of the figures assume 300MHz) and a blocking factor $m = 2$, hence memory access is uniform with a blocking time of $t^i_{block} = i$, and are presented in Table 3.

This table describes the WCET for a varying number of processors, and hence for different tree depths. It compares the worst case for a TDM-based system with that for the memory tree, but also outlines the WCET for the tree if certain levels are not congested. As an example, a tree for two clients has only a single multiplexor. In this case, the *L1* figure shows the WCET when this multiplexor at level 1 is congested, whereas *No blocking* shows the case where a request can be forwarded directly to memory with no blocking. Similarly, for four clients, two multiplexors are required. The figure for *L2* then shows the case for a fully congested tree, while the figure for *L1* shows the case where only the multiplexor at level 1 is fully congested.

The *TDM* case concerns itself with the worst case for TDM, assuming a step interval of 28 cycles (matching the response time of memory). This worst case will occur when the client in question issues a request just after it has been scheduled. In this case, it will have to wait for the intervals of all other clients to elapse (hence wait for $(nProcs - 1) \times 28$ cycles), and for the pipeline delay to elapse (hence another 25 cycles), and finally for all multiplexor transit times to have been satisfied (adding 9 cycles per tree level). It can then be seen that, while the worst case delay is worse than the TDM case for a fully backlogged tree, the tree will typically perform better when sections of the tree are not fully loaded.

Table 3: WCET (cycles) of a transaction for multiple processors.

Procs	*TDM*	*L6*	*L5*	*L4*	*L3*	*L2*	*L1*	*No blocking*
2	90	N/A	N/A	N/A	N/A	N/A	90	62
4	137	N/A	N/A	N/A	N/A	183	99	71
8	249	N/A	N/A	N/A	388	192	108	80
16	473	N/A	N/A	873	397	201	117	89
32	921	N/A	1750	882	406	210	126	98
64	1817	3523	1759	891	415	219	135	107

7 Requirements

This section lists the requirements which are of aspect CORE and scope NEAR from Deliverable D 1.1 that are relevant for the memory interconnect, prefetcher, and combined system listed in this document. The requirements for the memory controller are omitted and are listed in D 4.4. NON-CORE and FAR requirements are also omitted from this document.

S-0-501 Shared Memory

The platform shall provide means to share memory between applications and threads

The memory tree interconnect allows for multiple processors to access memory in a scaleable and analysable way.

N-2-011 No Re-Order

The processor may hgave several read or write requests outstanding. The NoC shall not reorder these read or write requests from the processor.

The memory tree interconnect preserves the FIFO ordering of requests from the processor, and does not attempt to reorder these requests.

N-3-047 Connection to Main Memory

The NoC shall provide the processing nodes with mechanisms for pulling data from main memory.

See S-0-501

M-5-064 Memory Access Latency

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

Both the memory controller and memory interconnect are time predictable, where a full system analysis of the required time bounds is presented within this document.

N-6-040/M-6-041 External Resource Access Latency

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

See above

References

[1] JEDEC Solid State Technology Association. *DDR3 SDRAM Specification*, jesd79-3e edition, 2010.

[2] NP Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373. IEEE Comput. Soc. Press, 1990.

[3] Micron Tech. Inc. *1Gb: X4, X8, X16 DDR3 Datasheet*, 2010.