T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 4.6 Final Benchmarking and Report of Memory Controller

Version 1.0
8 September 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**

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

Document Control

Version	Status	Date
0.1	First outline	17 June 2014
0.9	First draft	28 August 2014
1.0	Final version	8 September 2014

Executive Summary

This document describes deliverable *D 4.6 Final Benchmarking and Report of Memory Controller* of work package 4 of the T-CREST project, due 36 months after project start as stated in the Description of Work.

The document is organized as follows. First in Section 1, an overview of the memory subsystem is presented and the architecture of its constituent components, the memory controller, the Bluetree memory interconnect, and the prefetcher are briefly discussed. Four experiments are then provided in Section 2 to evaluate the memory subsystem. We first show that it operates in a functionally correct manner and that data written to the memory and is correctly read back. The second experiment demonstrates the reconfiguration features of the memory controller by showing that applications can be started and stopped dynamically, while other applications execute in a time-predictable manner. The third experiment evaluates the dynamically scheduled back-end and shows that it outperforms a state-of-the-art approach based on semi-static memory patterns both in terms of worst-case and average-case performance. The last experiment then shows that the prefetcher can be used to increase average-case performance without negatively impacting the worst case. The document is concluded in Section 3 by relating the memory subsystem to its specified requirements.

1 Memory Subsystem Architecture

This section outlines the architecture of the memory controller, the Bluetree memory interconnect, and the prefetcher, collectively referred to as the *memory subsystem*. Much of this overview is reused from previous deliverables, but is included to make this document self-contained and enable us to make specific points relevant to the final evaluation.

The architecture is presented in a bottom-up manner, starting with the SDRAM memory and the PHY in Section 1.1, the dynamically scheduled back-end (D 4.4) in Section 1.2, and the reconfigurable front-end (D 4.3) in Section 1.3. Together, these components comprise the *memory controller*, whose architecture is shown in Figure 1. After discussing the components of the memory controller, Section 1.4 presents the Bluetree memory tree that connects the memory controller to the Patmos processor tiles. The prefetcher is then discussed in Section 1.5, before the section is concluded in Section 1.6 with a brief explanation of how the memory subsystem has been integrated (D 4.5).

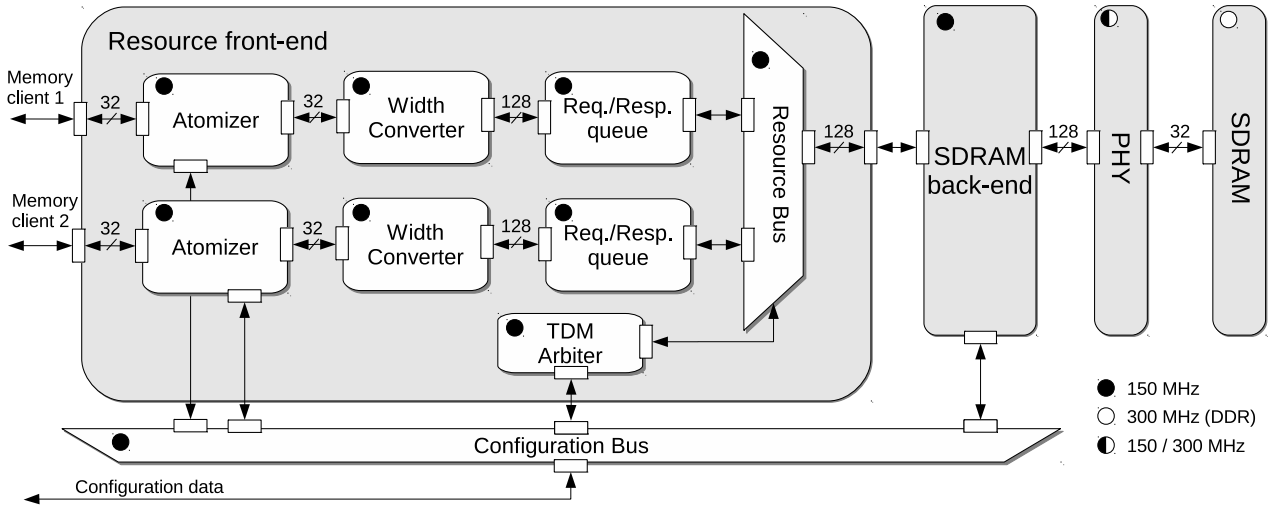

Figure 1: Memory controller architecture.

1.1 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 [10] 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 of approximately 3.3 ns.

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.

1.2 Dynamically Scheduled Controller 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 between the commands are satisfied. The 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, Stage 1 obtains the relevant information, including the transaction type (read or write), logical address, and the required number of bursts for the given transaction size. Stage 2 splits the transaction into the required number of bursts, and translates the target address to the corresponding physical address (bank, row and column) in the memory. Thereafter, the *command generator* in Stage 3 produces the required activate, read, write, and precharge commands for each burst. To prevent the memory from losing data, refresh commands are also generated periodically every 7.8 μs. 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. The *command scheduler* issues the command in Stage 5 if the timing constraints are satisfied. The action in each stage consumes one clock cycle, except burst splitting in Stage 2, which requires one cycle per burst, and command generation in Stage 3, which takes one cycle per command.

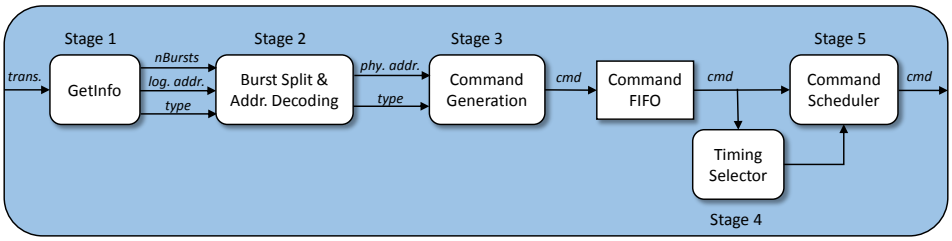

Figure 2: Architecture of the dynamically scheduled back-end.

The back-end runs at a frequency of 150 MHz, while the memory itself runs at 300 MHz. This means that the data path of the memory controller works with 128 B words provided by the memory tree internally, but reads or writes 4x 32-bit words per clock cycle to the memory to compensate for the double frequency (150 MHz to 300 MHz) and the double data rate (two data transfers per 300 MHz clock cycle).

1.3 Reconfigurable Controller Front-end

The primary function of the 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 behavior. 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, depending on the memory and the controller configuration. The configuration port on the atomizer allows the access granularity to be reconfigured at run time. The *width converter* accepts messages using 32-bit data words from the memory clients, 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), or 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 [7]. 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.

1.4 Bluetree

The Bluetree memory tree is a scalable interconnect designed to connect the Patmos processors to the memory controller, and is fully described and analyzed in D 4.5. The motivation behind a separate memory interconnect is twofold. Firstly, the inter-process communication requirements for a task and its memory requirements may not be similar, and hence the corresponding interconnects should be decoupled. Secondly, the tree system overcomes the problems with standard monolithic arbitration; distributing the arbitration throughout a tree can yield faster and more flexible systems.

Bluetree is made up of a number of 2-into-1 multiplexers. In the case where there is a valid packet at both inputs, the left-hand input takes precedence, although this behavior is configurable. Since the leftmost client has the highest priority, it is possible for this client to be able to dominate the tree and cause starvation of the rightmost client(s). In order to alleviate this issue, each router implements a simple arbitration policy, which tracks the number of times that a packet has been blocked by the left-hand side. When this counter reaches a fixed value, the right-hand side is given priority over the left for a single packet. This ensures that the tree is time-predictable and thus, coupled with the time-predictable memory controller, that the whole memory subsystem is time-predictable.

1.5 Prefetcher

The prefetcher is a component in the platform which attempts to improve the average-case execution time by speculatively issuing memory requests on behalf of processors. However, given the goals of the T-CREST project, the prefetcher *must not* cause a detriment to the worst-case timings of the platform.

The implemented adaptive-control scheme operates on the concept of a *prefetch slot*. This is a special blank Bluetree packet which can be filled in by the prefetcher, and is created in the system when there is slack time (or rather, when a packet is not present where one was expected). These can be created from two locations; either inside Bluetree multiplexers or from the cache that data was loaded into.

Recalling from Section 1.4 that each Bluetree multiplexer contains a blocking counter, which allows a low-priority packet a cycle of service when it has been blocked by m high-priority packets, this slack time can be created in two ways. A low-priority packet may gain service *without* being blocked by any high-priority packets, or $m + 1$ high-priority packets may gain service without being blocked

by any low-priority packets. Instead of using this blocking counter to log how many times a low-priority packet has been blocked, it can instead run all the time, dispatching a slot when there is no high-priority packet and $m > 0$, and also when there is no low-priority packet and $m = 0$.

The second location that these can be generated is from the target cache. In this case, when there is a prefetch hit, i.e. a prefetched line was useful, there will no longer be a demand access for the prefetched data. In this case, a prefetched slot can be dispatched to denote the "missing" memory access. This mechanism has been implemented into a custom "prefetch cache". This is a simple direct-mapped cache which is designed to be inserted in-between a processor and the first Bluetree multiplexer. This cache intercepts prefetches, relaying all other packets to the processor, and means that a prefetch mechanism can be implemented without modifying the target processor, since most cache systems do not allow data to be pushed into them from an external source.

This prefetch mechanism then can fit into the standard system analysis without any modification. Since the worst-case blocking calculation (e.g. that provided for Bluetree in D 4.5) must allow a request that has been admitted into the memory controller to complete, and also for all other higher-priority clients to deplete their bandwidth allocation, this prefetch mechanism is sound. Slots being generated on a prefetch hit replaces traffic that the processor would be generating without the prefetcher being present, and slots generated from the multiplexers represent clients not fully utilizing their bandwidth allocation.

1.6 Integration

As previously mentioned in D 4.5, the original intention was to connect the Patmos processors directly to the ports on the memory controller and to use the front-end to arbitrate between the different memory clients. While this approach is suitable for multi-core platform, it did not scale to the many-core T-CREST platform with a target of 64 Patmos processors. This is primarily an issue related to the scalability of the Resource Bus in the memory controller, whose centralized implementation cannot multiplex such a large number of clients at sufficiently high frequencies. Instead, the Bluetree memory tree was designed and implemented with a distributed implementation to replace the front-end. As seen in Figure 3, Bluetree connects directly to the memory controller back-end and the front-end is not used. In this deliverable, the front-end is only used to demonstrate and evaluate features unique to this component that are not available in the standard T-CREST platform. These features are still considered relevant to the project as they are available in platform instances with less processors, suitable for prototyping on common FPGA boards like the ML-605.

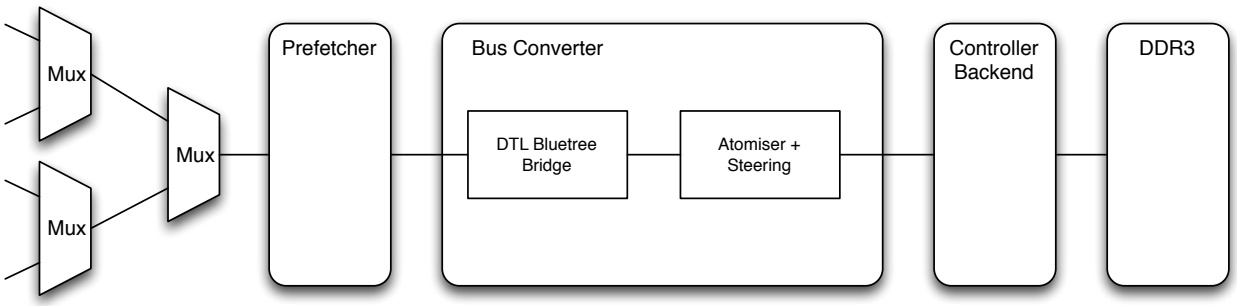

Figure 3: Integration between the memory tree and the memory controller

2 Evaluation

Having explained the basic architecture of the components in the memory subsystem, this section outlines the methodology used to evaluate each component and demonstrate that the combined system meets the requirements outlined in Section 3. While the benchmarking of the integration is new work, much of the work with research significance is adapted from previous experiments in published conference proceedings.

The rest of this section is structured as follows. Section 2.1 demonstrates the functional correctness of the memory controller and its interconnect. Section 2.2 shows the dynamic reconfiguration features of the memory controller, while Section 2.3 evaluates the dynamically scheduled controller back-end. Finally, Section 2.4 demonstrates that adaptive control of the memory interconnect can be used to improve the average-case execution time of tasks, while still maintaining the worst case.

2.1 Functional Correctness

This section evaluates the functional correctness of the memory subsystem. Functional correctness from the perspective of the memory subsystem depends on two axioms; any data written to memory must be readable again when accessing the same location, and that multiple clients accessing different memory locations must not interfere with one another. In order to demonstrate these axioms, we must assume that each core is utilizing mutually exclusive memory areas, since there is inherently a race condition where two cores are writing to the same memory location.

We demonstrate functional correctness using a multi-core memory test showing that cores can successfully write to memory and read back consistent data. This application was run on a 16-core instance of the T-CREST platform, utilizing the Bluetree memory interconnect and the dynamically scheduled controller back-end on a Xilinx ML-605 evaluation board. Within this system, the Patmos cores are clocked at 75 MHz, while the memory controller is clocked at 150/300 MHz. This test is written in C and compiled using the Patmos LLVM tool-chain. The resulting binary is then downloaded into the platform's external memory using the standard Patmos boot loader. After downloading, the master processor (by convention, CPU 0) then signals all other processors to start executing the same application.

The memory test first divides an area of memory into equal sized chunks for each processor. A barrier synchronizer is then used to ensure that all processors start at the same time. Each processor writes random data from a deterministic pseudo-random number generator (PRNG) to memory, each using a different random seed. It then resets the PRNG seed and reads the same data back, comparing it to the data that is re-generated from the PRNG. Each processor then writes its status to a status table in memory, and waits on a second barrier synchronizer. After crossing the barrier, the master processor checks that each processor was successful, or reports an error code to denote failure. This process is repeated a number of times with different random seeds.

Since Patmos contains a data cache, it must be ensured that the task is actually accessing memory, rather than simply reading from the data cache. For this reason, the data area is declared to be uncached, which causes the compiler to issue non-cached reads and writes (i.e. the `lwm`/`swm` opcodes rather than `lwc`/`swc` opcodes).

This memory test was executed on the T-CREST platform. In all cases, all cores reported correct results over all iterations of the memory test. This shows that multiple cores can access memory in a coherent manner, and that processors accessing memory do not interfere with one another. *From this experiment, we conclude that both the memory controller and Bluetree are operating in a functionally correct manner.*

2.2 Dynamic Reconfiguration

This section demonstrates dynamic starting, stopping, and reconfiguration applications in the memory controller, while other applications execute in a time-predictable manner. This experiment is summarized from [6], which explains the reconfiguration capabilities of the memory controller in detail. Note that this work uses an earlier implementation of the memory controller back-end based on hybrid scheduling with semi-static memory patterns instead of the dynamically scheduled back-end developed in T-CREST. However, this does not matter for the purpose of this experiment as the reconfiguration features are located in the memory controller front-end. In particular, it relates to the TDM arbiter controlling the Resource Bus in the front-end and its configuration infrastructure (see Figure 1), as well as our proposed time-predictable reconfiguration mechanism [6].

Seven synthetic applications (A-G) share the SDRAM memory in this experiment. Figure 4 shows the properties of the applications and the different use-cases that are traversed. Three maximum cliques (overlapping sets of applications) are identified, annotated with U1 (A, B, C, D), U2 (A, D, F, G) and U3 (A, E, F, G), respectively. At T1 and T2, applications are dynamically started or stopped, triggering use-case transitions that change the active maximum clique. The applications are implemented by four traffic generators connected to four ports on the memory controller front-end (applications on the same horizontal line are mutually exclusive and share a memory port and traffic generator). Figure 4 also shows the applications' bandwidth and service requirements. There are two classes of service requirements: 1) predictable service, which means that timing requirements on worst-case response time (WCRT) must be satisfied, and 2) composable service, meaning that the application must be temporally isolated from others in terms of actual execution time. For the purpose of this experiment, the former means that it is possible to dynamically change the TDM slot of the application as long as the WCRT remains valid, while the latter means that the application must use the same TDM slots throughout its entire life-time. For simplicity of the experiment, we assume all applications have a relaxed WCRT requirement of 2000 ns and issues only 512-byte requests.

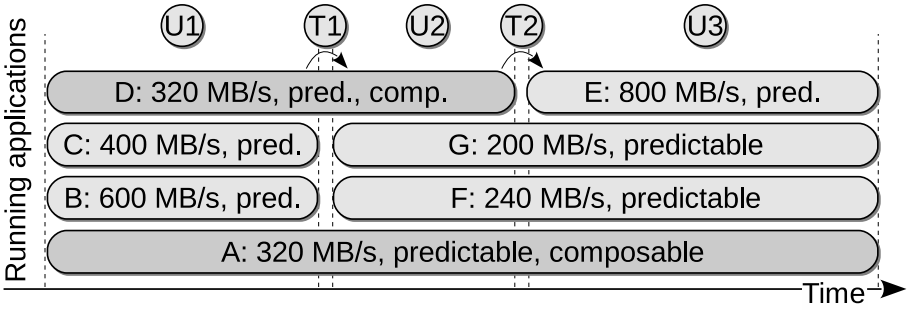

Figure 4: Temporal use-case transition behavior.

The memory controller back-end is configured with memory patterns that guarantees a worst-case bandwidth of 1862 MB/s. The TDM slot table size is set to 20, such that each slot corresponds to $1862/20 = 93$ MB/s. Because applications A and D require composable service, they get the same slot allocation in all use-cases where they are active. This is reflected in the allocation algorithms' output, which is shown in Figure 5. Note that without the proposed reconfiguration support, the slots for application F and G would not be movable, and application E would be unmappable due to fragmentation. This demonstrates the benefits of the reconfiguration capabilities.

U1:	A	A	A	A	D	D	D	D	B	B	B	B	B	B	B	C	C	C	C	C
U2:	A	A	A	A	D	D	D	D	F	F	F	G	G	G						
U3:	A	A	A	A	F	F	F	G	G	G	E	E	E	E	E	E	E	E	E	

Figure 5: TDM slot allocation results.

Figure 6 shows the temporal behavior of application F in two separate experiments. Each request is chopped into four atoms, and each bar represents an atom, which explains the periodicity. The bar height shows the actual measured response time.

In the first experiment (green markers), the safe reconfiguration mechanism in the TDM arbiter is switched off. At 68 μs, the reconfiguration from U2 to U3 takes place. The WCRT bounds of some atoms are violated as a consequence of reconfiguring the arbiter, which is unacceptable.

A second experiment (blue bars) is performed with the safe reconfiguration mechanism switched on. Here the WCRT bound is valid, and the actual response time is slightly lower during the reconfiguration process. *This experiment shows that our reconfiguration mechanism enables applications to be dynamically started and stopped while other applications execute in a time-predictable manner.*

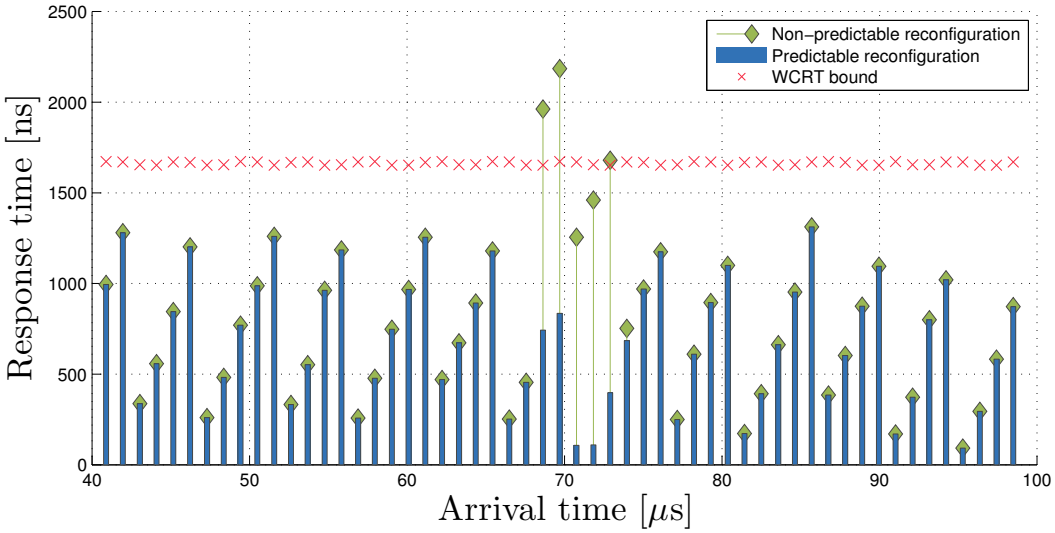

Figure 6: Response times with and without time-predictable reconfiguration.

To evaluate the reconfiguration time, we need to consider which parts of the memory controller have to be reconfigured. Figure 1 shows the configuration bus and the blocks that are connected to it,

which are the atomizer, TDM arbiter and the SDRAM controller back-end. Two scenarios can be distinguished:

1. A partial reconfiguration, where the memory controller remains active, but the arbiter is reconfigured. The experiments that were described in this section fall under this scenario. In this case only the allocation in the arbiter has to change.
2. A full reconfiguration of a quiescent memory controller, involving the atomizers, arbiter and controller back-end.

We discuss the reconfiguration time of each component separately, and later show how to determine the total worst-case reconfiguration time in the two scenarios.

The atomizer has a dedicated configuration port, and requires a single configuration word to be reconfigured. It takes one cycle to receive a configuration message, and another to bring the configuration change into effect. The reconfiguration time of one atomizer is hence two clock cycles, or 13.3 ns at 150 MHz.

The arbiter also has a dedicated configuration port. Each block of contiguous slots in the TDM slot table is encoded in a single configuration message. It again takes one cycle to receive the message, and another to commit the changes to the slot table. Hence, it takes 13.3 ns to reconfigure a contiguous block of slots into the arbiter.

The back-end has set of registers in which it stores the address decoder settings and the refresh interval. There are five registers in total, and each can be reprogrammed in a single cycle through a configuration message. It hence takes $5 \cdot 9.4$ ns to reprogram the back-end.

In a partial reconfiguration, the total reconfiguration time is given by the product of the number of contiguous slots that have to be programmed into the arbiter. Assuming a fine-grained TDM table of 64 slots, which all need to be reassigned to a different application, this takes $64 \cdot 13.3 = 853.3$ ns.

A full reconfiguration is similar to a partial reconfiguration with the additional cost of reconfiguring the atomizers and the back-end. We again assume a large system with 64 atomizers. The total time to reconfigure 64 atomizers is $64 \cdot 13.3 = 853.3$ ns. Adding the time required for the arbiter and back-end yields a total reconfiguration time of 1740 ns. *This shows that the reconfiguration time of the memory controller is well below the goal of 5 μs, stated at the start of the project.*

2.3 Time-predictable Dynamically Scheduled Back-end

This experiment demonstrates time-predictable execution of a dynamically scheduled DRAM controller and show advantages of hybrid scheduling using semi-static memory patterns. The experiment is summarized from [9], which presents the architecture, scheduling algorithm, and analysis methods in detail.

First, we evaluate our approach for requests with fixed size. Four memory clients are used, corresponding to four processors executing different Mediabench [8] applications (*gsmdecode*, *epic*, *unepic* and *jpegencode*). The experiment is executed for three different request sizes of 32 B, 64 B and 128 B. The results are compared to the hybrid scheduling approach based on semi-static memory

patterns in [2]. Two analysis approaches for our proposed controller are evaluated in the experiment: 1) The *analytical* approach assumes the worst-case initial state in all memory banks and computes the execution time (ET) of the request by determining the earliest time each DRAM command can be scheduled by the scheduler, considering its dependencies. If two commands can be scheduled in the same cycle (i.e. a read/write and an activate), it pessimistically assumes that postponing the activate by one cycle results in a corresponding increase of the ET. 2) The *scheduled* approach is less pessimistic, as it takes the initial bank state and the sequence of DRAM commands as input and actually runs the scheduling algorithm to determine the ET. This enables it to detect if postponing activate commands actually increases the ET or if this increase is hidden.

Figure 7 illustrates the WCET for the memory using different fixed request sizes. We can observe that: 1) the maximum measured WCET from the experiments is equal to the scheduled WCET bound. This indicates that the proposed analysis provides an exact WCET bound; 2) the scheduled WCET is never larger than the bound provided by the hybrid approach. *This suggests our dynamic command scheduling performs at least as well as hybrid scheduling in the worst case*; 3) the analytical WCET is equal to or slightly larger than the scheduled WCET. This difference is because the pessimistic assumptions about the scheduling of activate commands mentioned previously.

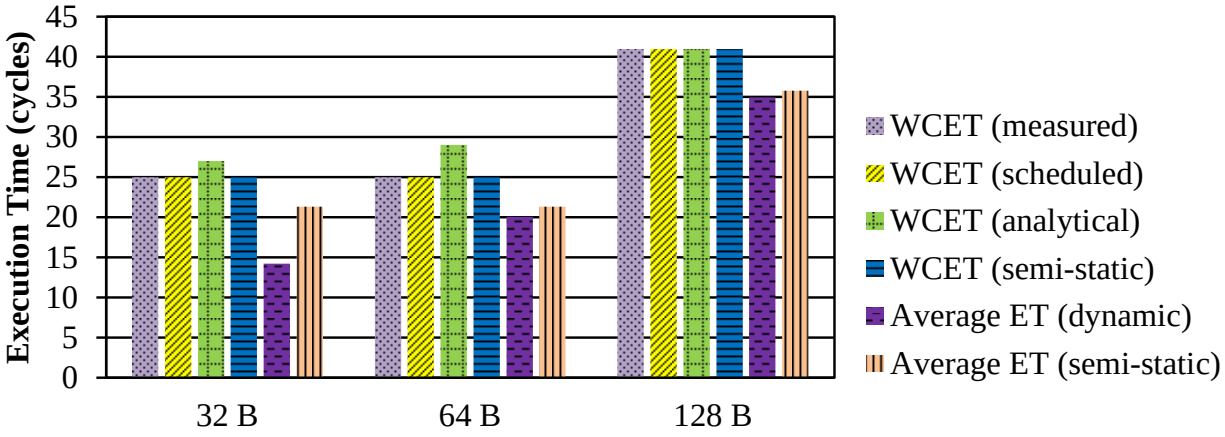

Figure 7: WCET and average ET of dynamic command scheduling and hybrid scheduling for fixed request size.

The average ET of dynamic command scheduling and hybrid (semi-static) scheduling are also shown in Figure 7. For each request size, dynamic command scheduling achieves lower average ET. This is because dynamic command scheduling monitors the actual state of the required banks and can issue commands earlier for a request that requires a different set of banks from that of the previous request. In contrast, hybrid scheduling [2] uses pre-computed schedules that assume worst-case initial bank state for every request. Figure 7 also shows that smaller requests benefit more from dynamic command scheduling. For example, 32 B requests gain 33.4% while 128 B requests only gain 2.3%. The reason is that smaller requests require fewer banks, which increases the probability that the following request accesses an independent set of banks and can thus be scheduled earlier.

Next, we evaluate our dynamic command scheduling approach with variable request sizes. A processor executes the *jpegdecode* application from MediaBench with a request size of 64 B, while the other components are represented by synthetic traffic generators with request sizes of 16 B, 32 B and

128 B, respectively. Figure 8 illustrates the WCET of dynamic command scheduling with variable requests sizes. As shown, the measured WCET is equal to the scheduled WCET, implying that the bound is exact. The analytical WCET is again slightly lower as it pessimistically assumes a command collision for every activate command.

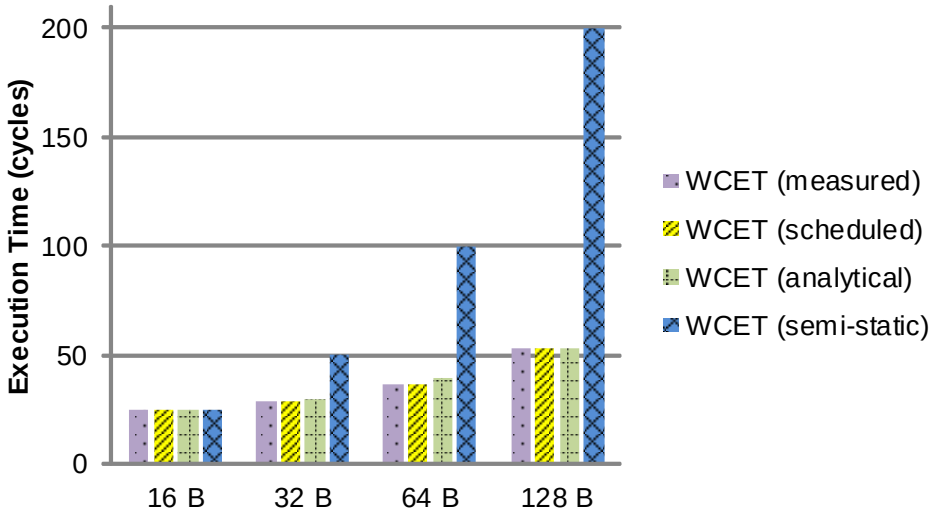

Figure 8: WCET and average ET for variable request sizes.

Note that we cannot elaborately compare our approach to other approaches in this experiment, as no other memory controller, including [2], provides an analysis that officially supports variable request sizes. However, a simple comparison can be made with the hybrid approach if we set the atom size in the atomizer equal to the smallest request in the system and that larger requests are served by scheduling many such smaller atoms consecutively. As seen in the figure, *the dynamic scheduling approach outperforms hybrid scheduling considerably, especially for clients with larger request sizes.* The reason is that small atoms are scheduled inefficiently to satisfy the timing constraints required to be able to schedule another atom to a different row in the same bank.

Another advantage of the dynamic scheduling approach is that it eliminates the reconfiguration overhead of the semi-static memory patterns. The dynamically scheduled memory controller back-end requires only five configuration words. In contrast, the hybrid scheduling controller needs to store all the command corresponding to the memory pattern sets it uses, each command using a single configuration word. Typically, this controller requires a pattern memory of 128 or 256 commands, and hence, the size of the configuration data has been reduced by 96% or 98% by switching to the dynamic scheduling approach. *This reduction of configuration data is significantly higher than the goal of 75%, stated at the start of the project.*

2.4 Adaptive Control

We proceed by demonstrating that adaptive control over the memory subsystem can be used to improve application performance at run time, without harming the worst-case response time. This section is summarized from [3], which provides a full timing analysis of the tree and an in-depth

variant of the evaluation summarized here. This work utilizes feedback from the memory tree in order to ascertain when there is spare time in the system, then utilizes this time in order to speculatively issue memory accesses on behalf of a processor. In effect, this work creates a prefetcher which can safely be used within real-time systems, and fit into the existing analysis for these systems. While this work has been evaluated using MicroBlaze processors, it is possible to apply the hardware and concepts to any processor architecture, including Patmos, since it does not require intrusive hardware modifications to the target processor.

The prefetcher has been built and evaluated using a subset of the TACLeBench suite of benchmarks [1]. These are a suite of benchmarks designed to benchmark WCET analysis tools, but chosen since there are no external library dependencies, and no external stimulus (or randomness) is used. This is beneficial for these experiments, since the timing behavior of the benchmarks *should* be identical on each run, and hence the effect of the prefetcher on the benchmark can be demonstrated.

The benchmarks have been executed on two hardware configurations; first, a "high-load" system with a single processor and fifteen hardware traffic generators. These traffic generators simply request from address zero on every cycle that their output queue is empty, and hence fill the traffic with a high amount of load in an attempt to simulate worst-case conditions on the tree. The second hardware configuration is one with sixteen processors, each running a copy of the benchmark.

A set of benchmarks were selected for use for this evaluation. This set of benchmarks was chosen for their behavior when interacting with main memory. They are typically streaming applications, accessing a sufficient amount of memory for a real evaluation. In addition, their streaming nature makes them ideal for evaluating a streaming prefetcher, which should be able to capture their streams and speed up their execution.

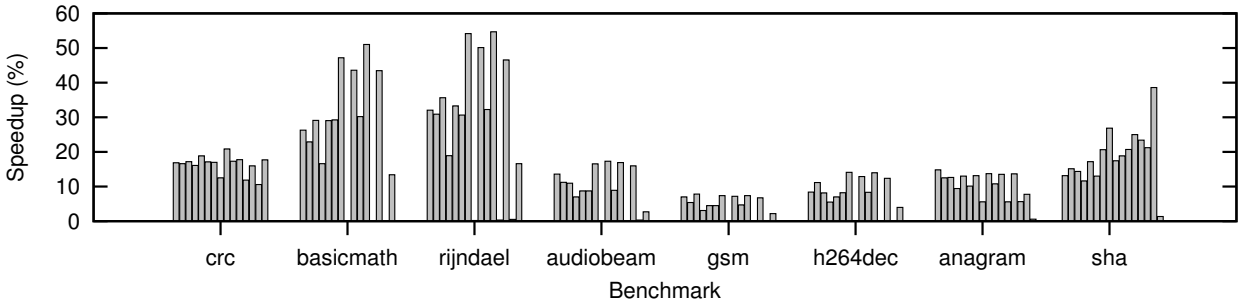

Figure 9: TACLeBench Results for one processor/fifteen hardware traffic generators.

Figure 9 shows the results of these benchmarks when run on the "high-load" system (i.e. a single processor with 15 hardware traffic generators). Here, we actually evaluate sixteen different systems, each with the processor being a different client on the tree (i.e. index 0 refers to the case where the processor is the leftmost (and thus highest priority) client on the tree, and index 15 refers to the rightmost position). The tree is configured with $m = 4$, simply to demonstrate the non-uniform nature of the tree. Here, speedups of 0-50% can be observed, depending upon the behavior of the benchmark in use.

Many of the smaller benchmarks operating on a stream of data, such as the *crc* and *sha* benchmarks yield a speedup between 0% and 40%. The reason for this is twofold; firstly, the code for these

benchmarks is reasonably small and so can fit entirely into the instruction cache. This allows the prefetcher to be able to operate solely on the data without any interference from code prefetches. In addition, there is enough computation for each cache line such that there is sufficient time to dispatch a prefetch.

basicmath_small and *rijndael_decoder* are examples where code prefetch is extremely effective. *basicmath_small* contains a math routine of size 2 KB, for example, which the prefetcher can fetch the entirety of accurately. *rijndael_decoder* contains a 8 KB block of straight-line code, which can again be accurately prefetched. The speedup for these benchmarks then depends upon the delay after which a prefetch slot can be generated, compared to the delay for which the benchmark would have to dispatch the read manually.

Other benchmarks show different levels of performance. *audiobeam* yields a speedup of 10-20% with the prefetcher enabled, simply because it operates on a stream of data, and that the computation time on this data is sufficiently long that a prefetch can be dispatched within the computation time. *gsm_decode* also operates on a stream of data, although the computation on this data is so long that the prefetcher cannot make a decent impact. There are some large decoding routines which can be prefetched though.

Another interesting note is the performance of the processors at indices 7, 11, 13 and 15. On many of the graphs, these processors show no performance increase with the prefetcher enabled. This is, in part, due to the slots mechanism mentioned earlier. These processors are in a position such that they are on the low-priority side of their first multiplexer. Due to the large amount of blocking that these cores exhibit, this means that it is likely that the processor requests the next required data before a prefetch slot has been dispatched. On other cores, it is more likely that a prefetch slot has been dispatched before the next datum is requested.

In these "high-load" systems, the presence of the prefetcher never causes a performance detriment; typically it can prefetch useful data, or is not able to fetch any data due to the load on the memory tree. These systems emulate the worst-case conditions on the tree, as all other processors are requesting data on every single cycle. *From this, we can conclude that adaptive control of the memory subsystem does not negatively impact worst-case performance.*

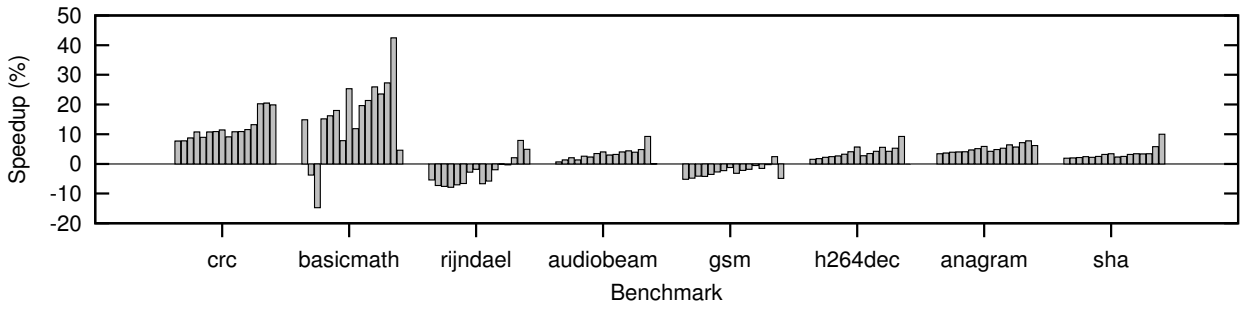

Figure 10: TACLeBench Results for the same task on sixteen processors.

The aforementioned benchmarks were also run on the sixteen processor system, all executing the same benchmark. The results of this can be found in Figure 10. Again, each bar refers to which processor index is being evaluated, with the left bar being processor index 0 (i.e. the highest priority), and the right being processor index 15 (i.e. the lowest priority).

The benchmarks yield speedups in this system too. All benchmarks tend to exhibit an upwards trend, with the lower-priority tasks benefiting more from the prefetcher being enabled. This is because with the prefetcher disabled, these tasks experience some blocking, but not as much as in the synthetic "full load" traffic generator systems. In most cases, the higher-priority tasks do not experience much blocking, and thus there is not as much latency for the prefetcher to hide. The low-priority tasks still experience a good amount of blocking however, hence there being more latency to potentially be hidden.

The *crc* and *basicmath_small* benchmarks yield a good performance improvement, for example. This is due to similar behavior to before; the prefetcher can easily prefetch across the whole stream of data, and as mentioned, the lower-priority tasks are favored since there is more latency to be hidden. *basicmath_small* yields a performance degradation on cores 1 and 2. Given the behavior of other cores, this is likely because prefetches get dispatched just before the processor dispatches a demand access for those lines, although fails to coalesce the prefetch and demand access, since there are still some cases for which this cannot happen.

For some tasks, such as *audiobeam* and *h264dec_ldecode*, their computation dominates over the memory delays that they experience. This effect is even more apparent in this system, again, due to the lower delay when communicating with memory. In the *rijndael_decoder* and *gsm_decode* benchmarks, the execution time is actually *increased*. This effect is due to two things when compared with the traffic generators; first, the time between requests is small (in the case of straight-line code, it is once every four cycles), and there is more contention over the amount of prefetches available. This leads to a situation where a prefetch may be dispatched too late, which ties up the memory controller with a useless access and thus causes a decrease in the average-case execution time. It is of note, however, that this slower execution time is still faster than those observed in the synthetic "full load" systems discussed previously.

From this, we can demonstrate that *adaptive control of the memory subsystem can be utilized to improve the average-case execution time of the system.* In addition to this, since prefetch "slots" are only created when there is an absence of memory traffic where the analysis would assume some memory traffic, we can analytically show that *adaptive control of the memory subsystem does not negatively impact the worst-case*, which is reinforced by the observed behavior of the "high-load" systems. A full description of this analytical approach can be found in [3].

3 Requirements

This section lists the requirements which are of aspect CORE and scope NEAR from D 1.1 that are relevant for the memory controller, memory interconnect, prefetcher, and combined system listed in this deliverable. 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 subsystem allows for multiple processors to access memory in a scaleable and analyzable way.

N-2-011/M-2-010 No Re-Order

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

The memory subsystem preserves 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

The memory subsystem is time-predictable. A full system analysis of the required time bounds has been presented in D 4.5.

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

M-2-012 Compare-and-Swap:

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

An SDRAM memory 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 implemented in the memory controller. Furthermore, time-predictable synchronization support for SDRAM controllers has already been investigated in [4, 5].

M-4-020 DRAM Configuration:

DRAM configuration is memory mapped within the memory controller.

The interesting configuration parameters in the reconfigurable memory controller 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 support for logging these using a cycle counter.

References

[1] TACLeBench, 2013. http://www.tacle.knossosnet.gr/activities/taclebench.

[2] Benny Akesson and Kees Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2011.

[3] Jamie Garside and Neil C Audsley. WCET Preserving Hardware Prefetch for Many-Core Real-Time Systems. In *Proc. 22nd International Conference on Real-Time Networks and Systems (RTNS)*, Versailles, 2014. To Appear.

[4] Mike Gerdes, Florian Kluge, Theo Ungerer, and Christine Rochange. The split-phase synchronisation technique: Reducing the pessimism in the wcet analysis of parallelised hard real-time programs. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 88–97. IEEE, 2012.

[5] Mike Gerdes, Florian Kluge, Theo Ungerer, Christine Rochange, and Pascal Sainrat. Time analysable synchronisation techniques for parallelised hard real-time applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 671–676. EDA Consortium, 2012.

[6] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens. A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, September 2013.

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

[8] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Int. Symp. Microarchitecture*, pages 330–335, 1997.

[9] Yonghui Li, Benny Akesson, and Kees Goossens. Dynamic command scheduling for real-time memory controllers. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

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