FPGA Implementation of a Time Predictable Memory Controller for a Chip-Multiprocessor System

Edgar Lakis

Kongens Lyngby 2013
IMM-M.Sc.-2013-1

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-1

Abstract (English)

The use of modern conventional architectures in real-time systems (RTS) requires complex analysis and suffers from high resource over-allocation needed to cover uncertainties stemming from employed speculative, average case optimizations. The design of time predictable RTS optimized architectures that allow easy timing analysis and tight timing guaranties is an active research topic.

The goal of this thesis is to explore the options for a predictable SDRAM controller for the T-CREST platform. The T-CREST project is an ongoing research project supported by the European Union's 7th Framework Programme, aiming to develop a homogeneous time-predictable multi-processor platform. The variable SDRAM access latencies pose some challenges for its effective use in RTS, while the many-core T-CREST platform creates a new context for rethinking the previous results and finding the new solutions for external memory access.

The simple working prototype of the single-port SDRAM controller is implemented and integrated with the processor. Several options for multi-port arbitration are considered, and proposal is made for arbitration and interconnect in T-CREST project. We evaluate our controller and make a closer look at one state of the art controller for RTS.

Preface

This thesis was carried out at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfillment of the requirements for acquiring an M.Sc. in Informatics.

The work was performed in the context of T-CREST project, supported by the European Union's 7th Framework Programme. The thesis was started on 1st of August 2012, were I have joined the T-CREST project team at the Technical University of Denmark. At the time of this thesis the T-CREST project was in the middle of its timeline. This posed additional difficulties, but all in all it was an exciting experience. From one side I had to catch up and fill the gaps of some preconditions that were decided beforehand. On the other hand the Patmos processor and the toolchain were still under development, and few times I had to hunt for the source of the problem in spectrum from compiler generated assembly down to VHDL code of the Patmos.

A month of my project time was spend on performing initial integration of Patmos processor with the early version of memory controller provided by the Technical University of Eindhoven, which is the T-CREST project partner responsible for delivering the final memory controller. The report of integration work is included in Appendix D.

Lyngby, 15-January-2013

Edgar Lakis

Acknowledgements

I would like to thank my supervisors and the rest of the T-CREST project team members at the Technical University of Denmark for the fruitful meetings. My supervisor Martin Schoeberl deserves a special thanks for his undisrupted availability in critical moments, irrespective of the geographical distance. I would also like to thank my co-supervisor Jens Sparsø for his coaching and for igniting my interest in digital design. It was after his course, few years ago, I started departing from my software oriented background. Many thanks go to Rasmus Bo Sørensen for valuable discussions and for his time spent proofreading.

I am very grateful to my family. Foremost, to my wife for her patience, support and her valuable time saving by helping to digitize the figures. I am also grateful to my sister for time spend with us and offloading part of the burden of being father and allowing me to spend more time on this work. Finally, I would like to thank my daughter, who seemed to grow together with this thesis, for her unreserved cheering attitude.

Contents

Chapter 1

Introduction

A real-time system (RTS) must perform its operation within a predefined time. For hard-RTS the delayed operation could cause serious consequences and must be avoided at all cost. Such systems must be rigorously analyzed and proved to always act on time.

To ensure the proper timing in all conditions, the timing analysis considers worst possible case. If the worst case performance is satisfactory, the system will to also operate properly under other conditions. Modern conventional architectures have a number of speculative features employed to improve the average performance. However the good average case performance have little advantage for hard-RTS, because the worst case operation has to be assured. To overcome the problems, predictable platforms are being designed. They improve the analyzability and providing better worst case performance guaranties.

The Synchronous Dynamic Random Access Memory (SDRAM) is widely used external memory because of its attractive combination of high capacity, low cost and competitive performance. However variable SDRAM access latencies pose some challenges for its effective use in RTS. The goal of this thesis is to explore the options for predictable SDRAM controller for use in the T-CREST platform. The T-CREST project is an ongoing research project supported by the European Union's 7th Framework Programme, aiming to develop a homogeneous time-predictable multi-processor platform.

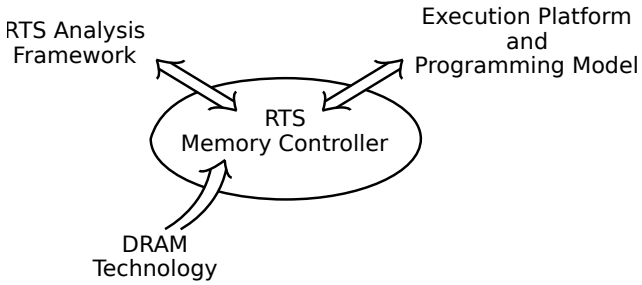

Figure 1.1: The context of the memory controller for the Real-Time System

Figure 1.1 presents the context of RTS memory controller design. First of all the knowledge of DRAM technology is needed because it is the source of the limitations. Secondly, the controller should interact well with the analysis framework, because it derives the performance guaranties for the whole system. Finally, the good performance guaranties of a memory controller are only possible by tuning it to the execution platform and programming model. Only by understanding the interaction of these three domains of knowledge the efficient controller is possible.

The thesis is organized in two parts. The first part provides the background and the context information for the work. Chapter 2 contains brief review of related work. Chapter 3 provides information on the DRAM Technology. Chapter 4 covers the relevant RTS notions. Its section 4.5 describes the T-CREST platform which is the context for the created controller.

Following chapters contain the main part of this work. Chapter 5 describes the design of the single-port memory controller which can also be used as a back-end for multi-port controller. Chapter 6 discusses the tradeoffs in multi-port controller design. The single-port controller is evaluated in Chapter 7. We also take a closer look at the state of the art multi-port controller. The thesis is finished with the conclusions in Chapter 8.

Chapter 2

Related Work

In this chapter we survey related work. First, we present previous works with the goal of creating SDRAM controller optimized for RTS. Next, we mention two publications regarding the SDRAM refresh. Finally, we list some works on memory access arbitration.

2.1 Controllers for Real-Time Systems

There are several previous works with the goal of creating SDRAM controller optimized for RTS.

Akesson et al. proposed the predictable controller called *Predator* [AGR07]. The work was later presented in more details in his PhD thesis [Ake10]. To have better data bus utilization, the large bank interleaved memory transfers are used. The closed page policy is employed, i.e. the banks are automatically precharged after each access. A number of interesting features were proposed.

The controller uses hybrid approach between the static SDRAM command scheduling which is good for known timing guaranties and the dynamic command scheduling with allows better average case memory utilization. The elementary operation size is fixed and the sequence of correctly interleaved SDRAM

commands to perform a read or write of elementary block are precomputed at design. The sequences are composed automatically by the computer program, which ensures that commands inside these patterns satisfy the SDRAM timing requirements. Auxiliary patters are also created to satisfying the timing during the changes in the transfer direction and to perform refresh.

The patterns and their compositions are analyzed at design time, to derive worst case memory access time (WCMAT) for each memory operation. The refresh is handled by including it in the WCMAT of each memory operation, but accounting for refresh period for transfers of larger sizes. To allow a better average case performance, the memory requests are translated dynamically into sequences of static patterns which are executed by the configurable SDRAM command generator. For hard-RT tasks all the responses are delayed to their worst case latency to provide an isolation of requestors' behavior.

The arbitration is performed at elementary block level, i.e. the larger transfers are always cut into elementary blocks and arbitration is performed for each block separately. Though any arbiter with bounded service time could be used by the framework, the *Credit-Controlled Static-Priority (CCSP)* arbiter is suggested. CCSP is a static priority arbiter augmented with rate regulators to limit requestors to their allocated bandwidth. The static priority makes decoupling of bandwidth and latency possible, allowing to give some higher priority requestors lower latency guaranties while leaving the bandwidth for other requestors. This however comes with the cost of significant increase in latency for lower priority requestors. We will look closer at the controller in this work. The arbitration is discussed in Section 6.2.5, while the synthesis results are presented in Section 7.2.

Paolieri et al. described the *Analyzable Memory Controller (AMC)* [PQCV09] which was part of MERASA project for predictable multi-core architectures. The bank interleaved command sequences are used as in the Predator, but the single set of sequences is created and analyzed manually whereas the Predator performs this by a program to allow exploring the latency/efficiency tradeoffs.

The fine grained Round-Robin (RR) arbitration is used for hard-RT tasks. They get higher priority than non-HRT tasks, which are scheduled in-order. The WCMAT is calculated by using the maximum possible time needed for a single transfer multiplied by possible number of colliding requesters (i.e. one pending non-HRT task and all other hard-RT tasks). Using the single maximum transfer increases the WCMAT beyond the value which is possible in worst case. For example, the worst case command is usually a Write invoked after Read. But it is impossible to have N Write-after-Read switches in sequence of N transfers. The improvements were proposed by the same authors in [PNC]. They accounted for maximum possible switches and additionally allowed preempting the non-

HRT transfer at the bank boundary, which allows saving few additional latency cycles.

The measurement based worst-case execution time (WCET) estimates for the tasks are used and refresh is handled by synchronising the start of the task with the refresh operation, to have refresh interference incorporated into WCET.

Reineke et al. describe the SDRAM controller for ARM based *precision timed architecture (PTARM)* processor [RLP+11]. The processor has four thread-interleaved pipeline, and assign separate banks for each thread, thus allowing use of banks without conflicts. Because such privatisation removes the concept of shared memory, the sharing is done through on-chip memory.

The refresh is performed manually to allow performing refresh in different banks independently. Because of their tight integration with the interleaved pipeline, they explore some properties of the architecture while issuing refresh. The refresh is deferred to the end of the read operation where it does not incur additional cost, because the pipeline can not utilize two consecutive read slots. For larger memory transfers (DMA transfers to/from the scratchpad) they account for maximal possible interference from refresh. Which takes to account how many periodic refresh operations are possible during the time needed to perform the whole transfer.

2.2 SDRAM Refresh

Atanassov and Puschner [AP01] described a problem with refresh incorporation into WCET of the task without considering each transfer. They showed, that even though the WCET augmented with the possible refresh interference is safe, the actual WCET path of the program might be different. This does not seem to be a problem in practice, if well behaved architecture without timing anomalies is used. The adapted adjustment formula is presented in Section 6.3.3.

Bhat and Mueller describe an approach of grouping the refresh operations together and executing them in separate special task [BM11]. This helps eliminate refresh interference uncertainty, because the refresh interference can be handled by the schedulability analysis. And the refresh operation does not have to be incorporated into WCMAT of each memory operation which is pessimistic. Unfortunately the authors did not mention that there are strong limitation on burst refresh in memories of later generations of SDRAM (Section 3.6.3).

2.3 Memory Arbitration for Real-Time Systems

Pitter has in part of his thesis performed some evaluation of arbitration schemes for RTS chip multiprocessor [Pit09]. The static (fixed) priority and time division multiplexing (TDM) arbitration were considered. The static priority without rate limiting was used, so only one higher priority requester could be incorporated, because more requesters could block the lower priority tasks indefinitely.

Puffitsch and Schoeberl [PS12] did some evaluation of TDM vs. RR arbitration when evaluating scalability of time predictable chip multiprocessor. They used a version of RR with one cycle empty slot per idle requester, because the RR which could skip all the idle requesters is not scalable.

Shah et al made few recent publications suggesting some arbitration schemes for shared resources in RTS and the SDRAM specifically. One proposal is *Priority Based Budget Scheduling (PBS)* [SRK12a], which as a name implies is fixed priority arbitration with rate limiting, similar to CCSP done in Predator. But the rate limiting is performed per time frame, whereas CCSP does it gradually. PBS grants each requester a memory access budget per certain time period, i.e. at the end of the replenishment period the budget of each requester is renewed.

Two hybrid arbitration schemes were also proposed by same authors. *Priority Division*, which is the TDM variation allowing to use idle slot a RR-like fashion [SRK11]. *Dynamic Priority Queue*, which is a RR with a per-requester budget limit for a replenishment period [SRK12b].

Chapter 3

DRAM Technology

This chapter provides the background on Dynamic Random Access Memory (DRAM) which will be needed in further discussion. For more detailed coverage of DRAM [JNW08] or [Wan05] can be consulted.

In this work, the focus is on Synchronous DRAM (SDRAM) which has been the most prevalent volatile off-chip memory for more than a decade. It is called synchronous to distinguish it from the asynchronous DRAM interfaces that were dominant at the time the standard was created. The standard is prepared by *JEDEC Solid State Technology Association* and several generations of the standards have evolved over the years. The following sections will describe the general features of SDRAM and the first generation in particular. The relevant changes of the later generations are mentioned in Section 3.5 section. Finally, the refresh operation is discussed in more details because of its implications for real-time system.

3.1 The Structure and Operation

The DRAM is called dynamic because the value of each bit is represented as a charge of a small capacitor, which discharges due to leakage over time. The

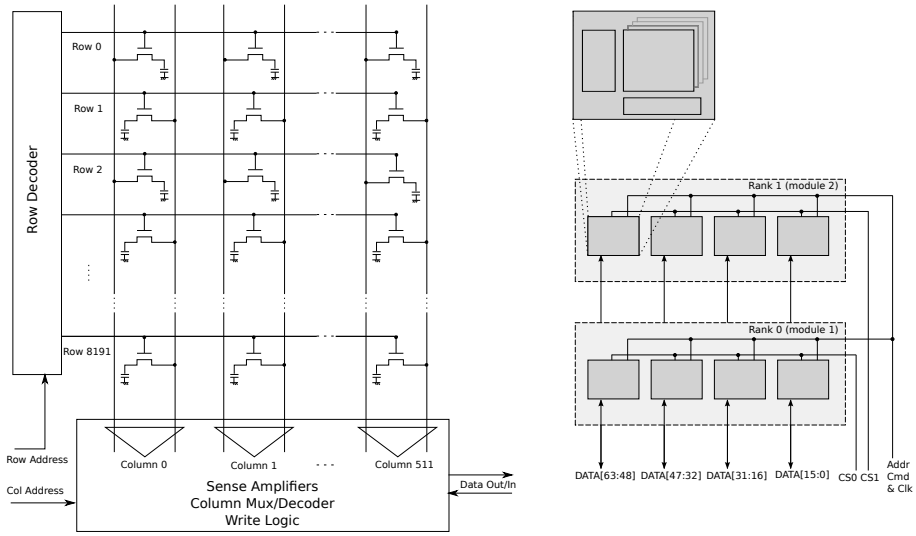

Figure 3.1: The example of four address dimensions of the SDRAM memory. *Left*: The schematics of the SDRAM array with addressable row and column; *Upper Right*: The SDRAM device with four banks; *Bottom Right*: Two ranks each in individual module composed of four devices operating on separate 16-bits of 64 bit data bus.

capacitors must to be refreshed periodically to preserve the valid value. In addition to the capacitor, the bit cell contains a pass transistor which is enabled when the value is read or written. The bits are not accessible individually, instead they are organized in arrays of $rows \times columns$. Figure 3.1 shows schematic drawing of 8192×512 array. A row must be prepared before its relevant bits can be read. This requires two steps:

precharge – the bit-lines (columns) are charged to midpoint voltage between logical 0 and 1.[1]

activate – the transistors of the single row are enabled, thus connecting the capacitors of the selected row to the bit-lines. The small charge of the capacitor creates small voltage swing on the bit-line. The sense amplifier recovers the value and drives the bit-line to ground or VDD voltage depending on the original value of the capacitor.

[1]The sense amplifiers are differencial, i.e. they detect the difference between two signals and need a reference voltage. To have more precise reference point for each column, the single column (single bit) uses two bit-lines. Where the half of the transistors are connected to each line. When the row is activated, the capasitor connects to one line, while the other acting as a reference

Both steps contribute significantly to the latency of the DRAM access, because the bit-line runs over the thousands of rows and has huge capacitance. But once the row has been activated its bits can be read/written with lower latency. During write, the new value is overdriven onto the appropriate bit-line and ends up in the capacitor.

Multiple arrays work in parallel with the same addresses to provide wider data words. But because the speed of the array degrades with its size, modern DRAM devices contain a third dimension called *bank* . The arrays of each bank can be used independently to increase the throughput[2]. This is performed by interleaving accesses to different banks, so that the data from one bank can be read/written while the other banks are busy while precharging or activating. The banks are still controlled through the same bus and usually share some hardware (Figure 3.1) so there are constraints on their usage.

Because the demand on the memory capacity is higher than what can fit on a single chip, multiple devices are combined on a PCB (Printed Circuit Board) *module* and operate side-by-side to provide wider data bus. The address space can also be increased, by connecting multiple devices to the same data bus and enabling single device by Chip Select signal depending on the address. This forth dimension is called *rank* (Figure 3.1). The modules have metal contacts on one edge and can be plugged into the special slots on the system mainboard.[3] Number of module configurations are possible, so each module contains the small non volatile memory which stores its timing parameters, which are read by the controller during the initialisation. This allows the controller configuring itself to satisfy the requirements of the module plugged into the system.

3.2 Signaling

The controller accesses the memory device through parallel buses, i.e. each signal bit is send independently on the same clock cycle. Conceptually following signal groups are used (with some modifications of later generations mentioned in later sections):

- Clock and command group:

[2]The name bank, can sometimes also be used to denote other things (see the rank, below). We use it only for independent portion of the SDRAM device, as used in the SDRAM device specifications

[3]The ranks are sometimes confused with banks, because some use the name bank for the socket in which the module is inserted

  - *CLK*, *CKE* and *CS#* (*#* denotes that signal is active low): clock, clock enable and chip select.
  - *RAS#*, *CAS#* and *WE#* : the encoded command. The signal names are from the time of asynchronous DRAM, but SDRAM samples all the signals at once and decodes them into appropriate command.
- Address group:
  - *BA<n>* : bank address, determines which bank of the device should perform the action.
  - *A<n>* : row/column address. The address bus is multiplexed, i.e. performs slightly different function depending on the command send through the control lines. The *A10* is used to enable the auto-precharge mode for the *Read* and *Write* commands (which don't need all the address bits to denote the column, because there are less columns than rows). The same *A10* pin is also used in *Precharge* command to specify that the operation should be applied on all banks. The address pins are also used to specify the value for the *Mode Register Set* command (see next section).
- Data group:
  - *DQ<n>* : bidirectional data lines.
  - *DQM* : data masking. Signal controls tristate output buffer during the *Read* and masks the input during the *Write*. The 16-bit wide devices use two separate mask signals (*UDQM* and *LDQM*) for independent control of the upper and the lower byte.

The data bus and the control/address bus are independent; this allows sending commands to a different bank during the longer data transfer cycles. As mentioned earlier, multiple devices can be used to provide wider data bus. In this case the control and address lines are connected to every device and data bus is sliced across devices. Alternatively multiple devices can be combined into a larger address space, by also sharing the data group signals and enabling the appropriate device with a chip select.

The is no handshaking/acknowledge signals in the interface, instead there are implicit timing parameters of the memory chip which must be obeyed by a controller to assure proper operation. The timing parameters are presented in Section 3.4.

3.3 Commands

The SDRAM has a synchronous interface and is operated by a predefined set of commands. Because only the bits of the active row are directly accessible,

in general case the read/write operations involve a sequence of SDRAM commands, also called transactions. In this section we will first look at the available commands and put them in context by few examples of simple transactions.

Three command signals (*RAS#*, *CAS#* and *WE#*) allow to represent 8 different commands. Additionally the value of clock enable signal (*CKE*) is used to enter the power saving modes but it is not discussed here. We now list the common commands (with their parameters specified in parenthesis) and the associated timing requirements, which are covered in more details in Section 3.4:

- *Precharge (bank/ALL)*: charges the bit-lines of the specified (or ALL) banks to reference voltage, to enable data recovery by the sense amplifiers during subsequent row activation. The precharge requires t_{RP} time before *Activate* command can be issued.
- *Activate (bank,row)*: Activates the row of the specified bank. The data of the activated row will be available for *Read*/*Write* after t_{RCD} time, but restoration of the values into the bit capacitors usually requires more time, specified as t_{RAS}. The bank can not be precharged before that.
- *Read (optional auto-precharge)*: Requests to read a number of words from the active row. The data can be sampled from the data bus in t_{CAC} cycles. The length of the transfer is configured by *Mode Register Set* command. The device will issue the *Precharge* command automatically at earliest allowed time if auto-precharge was enabled for this *Read* command. In addition to t_{RCD} mentioned in *Activate* command, there might be constraint on minimal separation between the consecutive *Read*/*Write* commands specified as t_{CCD}.
- *Write (optional auto-precharge)*: Requests to write a number of words into the active row. For the SDR SDRAM (see Section 3.5.1) the data burst is started together with the command, for later generations, the data must be delayed. The burst length is also controlled through *Mode Register Set* command, but can either be set to one or the value of the read size. The notes regarding the t_{CCD} , t_{RCD} and auto-precharge of *Read* apply to *Write* as well . With additional constraints (t_{WTR} and t_{WR}) required to allow the last data word to be stored properly.
- *Burst Stop*: Allows to interrupt the current transfer. The command does not have immediate effect and the transfer is interrupted after the t_{CAC} cycles. The command is slightly redundant (and in fact has been removed starting from DDR2 SDRAM), because the bursts can be interrupted by the *Read*,*Write* or *Precharge* commands.
- *Auto Refresh*: Performs parallel *Activate* followed by *Precharge* for all banks. The row address for the activation is supplied from the internal counter which points to next row after each refresh. The time needed to complete the refresh is specified as t_{RFC} which might be larger than

$t_{RAS} + t_{RP}$ required to do the same for single bank, because more current is needed. The t_{RFC} can also increase, because larger devices perform refreshes of several consecutive rows for single command.

- *Mode Register Set (reg,value)*: The command is used to set come configurable registers of the SDRAM device which alter its mode of operation. The most common parameters are the t_{CAC} cycles, burst sequence and length. The later generations have larger configuration and also use the command to perform some calibration. The command requires t_{MRD} time of subsequent inactivity.

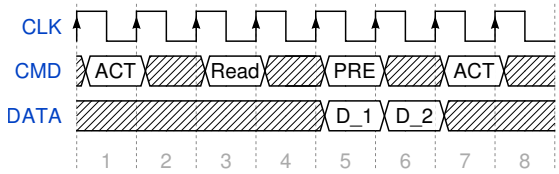

Figure 3.2: SDRAM(SDR) command sequence for burst read of two words. The closed page policy used, i.e. the bank is precharged after use.

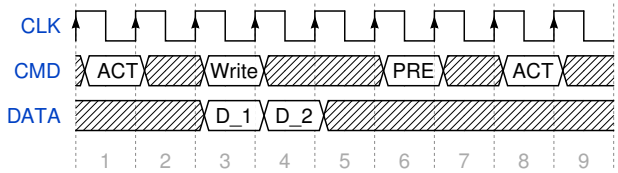

Figure 3.3: SDRAM(SDR) command sequence for burst write of two words.

Because the single *Read* and *Write* commands can only perform a transfer with the active row. In general case extra commands will need to be issued to perform an read or write operation. The Figure 3.2 and Figure 3.3 show the transactions for general two word transfer operations. In the examples the *Precharge* command is issued explicitly to show its place in the transaction. The action could also be scheduled for automatic execution by enabling the auto-precharge during *Read*/*Write* command.

3.4 Timing Parameters

The timing parameters of the SDRAM device need to be obeyed by the controller. Some parameters are the specification of the device behavior like t_{CAC}

describes in which cycle after the *Read* command the result would be available on the data bus, the others are requirements for the behavior of the controller. The parameters can be divided into two groups. The first group are the usual signal integrity requirements which are present when interfacing any synchronous component. Those are setup/hold requirements for the inputs with respect to clock/strobe signals. The clock-to-output timing are provided for all the outputs and the timing of the tristate buffers are needed for bidirectional signals. The other group are protocol level requirements which describe the separation between commands/data on the SDRAM bus which is needed to avoid hazards for shared hardware inside the memory device.

The use of single bidirectional data bus requires extra delay when the direction of the transfer is changed. The extra separation might also be required when getting data from the different ranks [JNW08]. The sharing of the I/O and control hardware by the banks requires delays between the commands even if they are directed to different banks.

3.4.1 Parameter List

This section has a brief description of common SDRAM timing parameters. The memory device specification usually specify these parameters in nanoseconds, but the synchronous memory controller issues the commands on clock edge, so the parameters need to be rounded to full clock cycles.

t_{RSC}**: Register Set Command** – the time needed to complete the *Mode Register Set* command delay.

t_{RP}**: Row Precharge time** – delay between *Precharge* and *Activate* command to the same bank.

t_{RRD}**: Row-to-Row Delay** – delay between *Activate* commands to different banks.

t_{RCD}**: Row-to-Column Delay** – delay between *Activate* and *Read* or *Write* command to the same bank (i.e. activated row).

t_{CCD}**: Column-to-Column Delay** – delay between two consecutive *Read* or *Write* commands.

t_{CAC}**: CAS Latency** – delay between *Read* command and output of first data word. For synchronous DRAM this value is always rounded to clock cycles, sometimes it is also denoted CL.

t_{RAS}**: Row Access Strobe** – delay between *Activate* and *Precharge* command to the same (bank). The delay is needed for sense amplifiers to charge the capacitors discharged during the activation. Care must be taken for this to hold also then the command is issued implicitly through auto precharge, i.e. the short read/write burst might need to be delayed after the activa-

tion.

t_{RC}**: Row Cycle** – delay between successive *Activate* and/or *Auto Refresh* commands.

t_{RFC}**: Refresh Cycle** – time needed for *Auto Refresh* command. All banks must be idle and not used during t_{RFC} (see: Section 3.6.2).

t_{PQL}**: Last Output Data to Precharge** – usually negative parameter specifying by how many cycles the *Precharge* can overlap with the ongoing read burst transfer. The later generations instead use t_{RTP}**: Read-to-Precharge** parameter for separation of *Precharge* after *Read*.

t_{DPL}**: Input Data to Precharge delay** – time from last data written on the bus to when *Precharge* can be issued. The alternative name t_{WR}**: Write Recovery** is used by later generations.

t_{WTR}**: Write-to-Read** – minimum separation for *Read* after the *Write* command introduced in later generations.

t_{WL}**: Write Latency** – rounded to cycles separation between the *Write* command and first data word introduced in DDR2. The DDR uses $^*t_{DQSS}$ parameter in ns and write latency of one cycle.

t_{FAW}**: Four Activate Window period** – this is constraint of 8 bank devices, which allows only 4 banks to be activated in the rolling t_{FAW} window. The 8 bank devices are possible for DDR2 and mandatory for later generations.

3.4.2 Parameter Relations

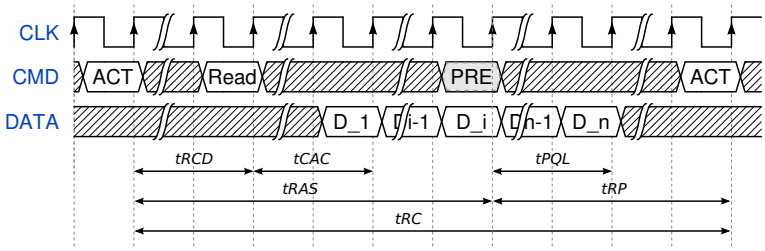

Figure 3.4: SDRAM(SDR) read transaction timing parameter relation. The *Precharge* command is marked gray, because it can be omitted if the auto-precharge was enabled during the *Read* command.

To summarize the timing parameters of the memory operations we show them in context of simple read (Figure 3.4) and write (Figure 3.5) bursts. The first SDRAM generation is used to make the examples simple. The closed page policy is used in the examples, i.e. the row is precharged after the *Read*/*Write* command to have lower latency for next access (assuming that they are random). The don't care parts of the command and/or data buses can be used by

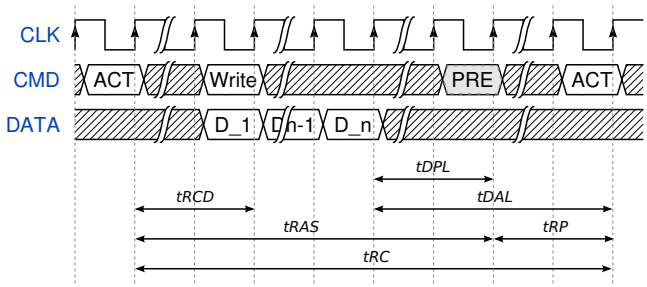

Figure 3.5: SDRAM(SDR) write transaction timing parameter relation. The *Precharge* command is marked gray, because it can be omitted if the auto-precharge was enabled during the *Write* command.

operations to other banks, if they satisfy the bus turn-around requirements for data direction change and the t_{CCD}, t_{RRD} and t_{WTR} constraints.

Few observations can be made here. The short burst of just one or two words will result in very low data bus utilization, because the length of whole transaction is limited by the t_{RAS} and t_{RC} (can not be shorter). For read transaction the *Precharge* command can be started while the rest of the data is received. While the write transaction requires additional delay (t_{DPL}) between the last data cycle and the *Precharge* because new values must propagate into the capacitors.

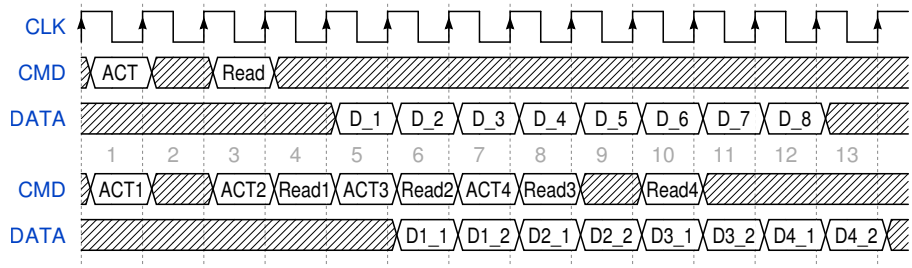

Figure 3.6: SDRAM(SDR) Long burst to the single bank (top) vs. short bursts interleaved across 4 banks (bottom).

The Figure 3.6 demonstrates the two possible ways of implementing longer transfers inside the controller. The first one is single read of 8 words from the same bank. The second uses four short bursts of 2 words interleaved onto four banks. The first is more energy efficient and in this example have one cycle shorter latency, but the second allows to use full memory bandwidth for back-to-back read or write operations. This is because the first approach uses the bank until

the end of the whole transfer in cycle 12. The subsequent read request for other row of the same bank would require at least $t_{RCD} + t_{CAC}$ cycles until the new data appears on the bus, and even more if bank precharge can not be fully overlapped with the data transfer (t_{PQL}). In the interleaved approach the first bank becomes available already in 8'th cycle, and can begin operation for the new read request while the data from other banks is transmitted. The examples of how the consecutive transactions are overlapped is presented in Section 5.3.3

3.5 SDRAM Device Standards

This section covers some of the features of different standards that might have effect on memory controllers. The main focus is on JEDEC SDRAM/DDR standards because of their wide use. We also limit ourselves to memory device standards. The memory module standards describe how devices are arranged into modules and how they should be wired, and though this degrades the timing, it does not change the principles of the operation.[4]

3.5.1 SDRAM: Synchronous Dynamic Random-Access Memory

This is the first SDRAM standard and is now referred as Single Data Rate (SDR) to distinguish it from later Double Data Rate (DDR) standards. Single command and/or data word is transferred in one clock cycle. Unfortunately the standards document does not seem to be publicly accessible, so a datasheets of specific chips were used when preparing this section.

The memory device might have 2 or 4 banks. The first devices were supporting the clock frequencies of 66 to 100 MHz but more recent 64 Mb chips can be operated at 200 MHz (while the 512 Mb parts support 133 MHz) [MT12]. Each frequency has a fixed range of supported t_{CAC}, and the controller picks one by configuring the device.

The number of word transfers used in single Read/Write operation is also runtime configurable, and can be 1, 2, 4, 8 or (optionally) whole row. The length of write burst can be either set to 1 or the length of the read burst. The longer burst can also be terminated by new *Read*, *Write*, *Precharge* or (optionally) explicit *Burst Stop* command. *Precharge* command only has a terminating effect if it is

[4] We do not consider the fully buffered modules or similiar buffers on board because they use different interface

issued to the bank that is currently performing read/write to allow concurrent control of the banks.

Two possible orderings of words inside the non burst-length aligned accesses are possible. In the *Sequential* ordering the addressed word is followed by those with subsequent addresses, until they wrap around on the alignment boundary (i.e. the last word before the boundary is followed by the words from the start of the addressed block). The *Interleaved* order is provided for simpler implementation of the caches. The address of the current word is calculated as $initial\_address \oplus words\_transmitted$.[5]

3.5.2 DDR SDRAM: Double Data Rate SDRAM

The next generation of the SDRAM doubled the possible bandwidth by transmitting two data items in one clock cycles. For this reason the second and the next generations are called Double Data Rate (DDR). The standard specification is freely available on JEDEC web site [JES08].

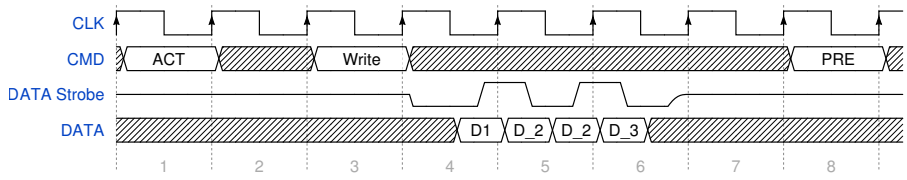

Figure 3.7: DDR example: Write transaction of 4 words

The doubling of the bandwidth was possible because the whole row of data is available after the activation. The I/O interface fetches twice the data from the memory core and transfers it in two half cycles. Because the possible window for stable data values was halved, the new bidirectional signal was introduced into the data bus. The new Data Strobe (DQS) signal is used for source synchronous transmission, i.e. acts as a clock which is sent together with the data. This way the jitter and the skew between the data and the "clock" is reduced. The data is center aligned to DQS during writes and is edge aligned during reads. A Delay Locked Loop is used on DDR chip to align the data/strobe with the original interface clock. The interface clock used for command and address bus became differential to reduce the jitter. The recent devices use the same frequencies (133 MHz to 200 MHz) as the SDR devices, but provide the double bandwidth and larger capacities [MT12].

[5] $\oplus$ symbol denotes xor (exclusive or) operation. The *words_transmitted* counts number of the words transmitted so far in the current burst

Because of the double prefetch performed from internal DRAM array, the single word read/writes are only possible by masking out the unused word in the pair. That is the supported burst lengths are 2, 4 or 8 words (the full row bursts were removed from the specification). According the specification, the DDR devices always contain four banks.

As for the timing parameters changes, the separation of the write command and write data is introduced. There are also requirements for the strobe signal relation to the data. The Figure 3.7 shows this on example write transaction with burst length of four. The data strobe signal must be low for some time before the data (preamble) and after the data (postamble).

3.5.3 DDR2

As the name implies the DDR2 [JES09] is successor to DDR standard which further doubled the bandwidth, by doubling the prefetch. The DDR2 internally fetches 4 words at a time from the memory core and than sends it at the faster rate. The data bus remained double rate, but the frequency of interface clock was doubled. Naturally the change eliminated the bursts of two words. Also the *Burst Stop* command was removed.

The introduced option for differential Data Strobe allows increasing the possible interface frequencies up to 533 MHz at the expense of longer t_{CAC}. The controller must dynamically control the On Die Termination parameters on the memory devices if multiple ranks are used. The device capacities increased by option of having 8 independent banks per device, but there is a limit on how densely in time the banks can be activated. The requirement is captured in t_{FAW} parameter which specifies the time window in which no more than 4 activates can be issued, i.e. the minimum separation between the i^{th} and the $(i+4)^{th}$ activate.

The DDR2 introduced *Posted CAS* also known as *Additive latency*, which simplifies the controller, by allowing it to reduce command bus conflicts. When the additive latency is enabled, the *Read/ Write* command can be issued in the next cycle after the respective *Activate* command. The hardware on the memory device would then internally delay the command for configured number of cycles to ensure that the t_{RCD} is satisfied.

3.5.4 DDR3

The DDR3 [JES12a] allows double memory throughput by doubling the internal DRAM core prefetch and the interface frequency. The burst length becomes fixed at 8 words, with the option to discard the half of it. The chopping of bursts to 4 words still requires the four cycles of separation between consecutive commands (equal to 8 transfers on the data bus). The separation is only reduced for write command after read.

The interface frequency is increased to up to 1066 MHz. With the increase of device capacity the 8 banks per device becomes a requirement. With the same limitation of bank activation frequency as for 8-bank DDR2 memories (t_{FAW} time for four activates), but with more acute effect. For high capacity devices the t_{FAW} (in cycles) is higher than 4*8, so it is not possible to fully utilize the data bus for random reads/writes.[6]

To allow such a high frequencies on the data bus, the fly-by connection of the clock/command/address signals to the modules is used. In previous generations, the clock/command/address lines to different chips on the module where delay matched to have low skew between different chips. In DDR3 they reach the chips in series, creating the skew between the data bits connected to different modules. The controller must deskew the signals by introducing additional delay on fast paths. The delay calibration is performed during the initialization and must also be performed during the runtime to accommodate the changes caused by temperature changes.

3.5.5 DDR4

The DDR4 [JES12b] is the latest generation of the DDR standard family publicly released on September 2012.

The DDR4 architecture uses an 8-n prefetching as the previous DDR3 generation, supporting the same burst length of 8 and burst chop of 4. The number of possible banks is increased to up to 16. But their concurrent use is constrained. The concept of bank groups is introduced with bigger timing constraints for the commands to the same bank group.

[6] The random reads/writes implemented by interleved access to consecutive banks. The back-to-back operations to the same opened row can of course have a valid data on the bus each cycle.

3.5.6 Wide IO SDR

Wide IO Single Data Rate (WIDE IO SDR) is JEDEC standard [JES11b] for future embedded systems. It targets the systems where one or more memory dies are stacked over the system die (including processors, caches, memory controller etc.) and connected by Through Silicon Vias (TSV). The 3D stacking loosens the I/O limit because dimensions of each I/O bump becomes smaller. This enables to switch from trend of increasing frequency as seen in evolution of DDR standards into increasing the parallelism. The memory chips are accessible through 4 independent interfaces with 128-bit wide data buses operating in SDR mode at 200 or 266 MHz.

3.5.7 Other Synchronous DRAM Interfaces

In addition to mentioned DDR generation family, the JEDEC defines the two related interfaces: Low Power Double Data Rate (LPDDR) and Graphics Double Data Rate (GDDR). The standards are based on their DDR counterparts, but the former has interface optimized for higher throughput and the former, as the name implies, for reducing power consumption.

Few non JEDEC interfaces are/were popular. The Reduced Latency Dynamic Random Access Memory (RLDRAM) is proprietary interface providing low read latency. The devices have 8 or 16 banks and additionally allow to trade-off the capacity for speed by enabling multi-bank write mode during which the same data is storred into 2 or 4 banks at the same time, allowing the duplicated data to be read from different banks without t_{RC} penalty. The row organization as well as Activates/Precharges are not visible to the user. The memory is accessed through simple Read/Write command.

Few other proprietary interface generations were designed by Rambus. The DRAM is accessed through narrower partly serialized interface where the commands and data is transmitted as a packets over several cycles. The serialisation of the interface can also be seen in some server computing where large address spaces are needed. Such systems would use hierarchy of memory controllers where custom narrow interface is used for inter controller communication and the standard JEDEC compliant interface is used to communicate with standard JEDEC modules [CBRJ12].

3.6 DRAM Refresh

The refresh of the capacitor charge is essential for correct operation of the DRAM. This section describes refresh operation, its requirements and refresh related limitations across SDRAM generations.

3.6.1 Different Ways of Performing Refresh

There is some flexibility in performing refresh. The methods of invoking a refresh are listed first, followed by the possible ways of organizing them in time.

Self Refresh This is autonomous refresh mode which can be performed by the SDRAM chip during longer inactivity periods. The chip has to be brought to this special power saving mode. Because it takes relatively long time to return to normal operation, such method is not relevant for further discussion.

Auto Refresh [7] The refresh is triggered by issuing dedicated refresh command. The command does not specify the address of the row; instead the row is pointed by the internal counter inside the chip. The same row is updated in all the banks in parallel, and the counter is incremented to point to the row for the next refresh operation.

Refresh by Activate The charge is restored by the sense amplifiers during the row activation. So activating a row has a side effect of refreshing (irrespective of presence of following Read/Write to this row). Activation is performed for a row of a single bank, so all the banks have to be refreshed separately. Additionally controller/software needs to keep track of the row counters.

Each row need to be refreshed within certain time period, and there are two strategies regarding the refresh of different rows:

Distributed Refresh This is the usual way. The refresh action are spread out evenly in time. For example, if DRAM contains 8192 rows, issuing Auto Refresh every 7810 ns assures that each row meets requirement of 64 ms refresh period ($7810 \times 8192 \approx 63.98 \times 10^6$ ns).

Burst Refresh Refresh actions occur in bursts. There is a longer period without refresh followed by several refresh operations invoked one after another.

[7] Also known as CBR (CAS Before RAS) from the times of non-synchronous DRAM interface

3.6.2 Refresh Timing

Even though retention times of different DRAM cells might differ by the orders of magnitude, the specification requires the same refresh period for all the rows, which is usually 64 ms. The different ways to exploit this fact have been proposed in the literature, but the subject is out of the scope of this work.

Table 3.1: The t_{RC} and t_{RFC} parameter values (in nanoseconds) from JEDEC specifications for some memory generations.

	t_{RC}	64Mb	256Mb	512Mb	1Gb	2Gb	4Gb	8Gb
DDR	55-70	70-80		70-80	120-130			
DDR2	55-65		75	105	127.5	195	327.5	
DDR3	43.3-52.5			90	110	160	260	350

The SDR SDRAM describes refresh requirement in terms of retention time t_{REF} for each cell (which is the same as each row). The specifications of subsequent generations use t_{REFI} parameter, which is longest period between two consecutive refreshes operations. For simple memories which perform a refresh of single row during one operation the relation between the two parameters is $t_{REFI} = t_{REF}/N_{rows}$. But bigger devices of later generations refresh multiple rows during the single Auto Refresh operation, hence the t_{REFI} parameter is given in the specifications. While the time needed by the Auto Refresh operation is given in t_{RFC} parameter. It is usually slightly greater than Row Cycle time (t_{RC}) for very small devices, but can be several times greater for the larger devices (see Table 3.1). The large t_{RFC} might constraint the memory scheduling scheme as discussed in Section 6.3.3.

The retention time is temperature dependent, because the leakage currents are larger in warmer chip. The specifications usually require doubling the refresh rate if the temperature is higher than $85°C$. This slightly reduces the memory throughput, but does not complicate the controller design.

3.6.3 Burst Refresh Support in SDRAM Generations

The first generation SDRAM (Single Data Rate) devices usually allow performing the Refresh command bursts of arbitrary length. Thus refresh actions for all the rows can be grouped together and invoked at convenient time to avoid refresh interference with normal operation.

The later, Double Data Rate generations only allow limited flexibility. The maximum of 8 Auto Refresh operations can be postponed or pulled (issued in advance), but not both. In another words, the maximum of $9 \times t_{REFI}$ interval between surrounding Auto Refresh commands is allowed. The limiting of refresh burst length is caused by the power requirements for the refresh operation. That increased with higher device densities of later generations. The reason for having maximum interval without refresh is less clear. The requirements is stated without any comments in DDR specification [JES08]. The specification of later generations state it more explicitly, but also without clarifications [JES09] [JES12a] [JES12b].

The limitation of DDRx memories does not affect bursts of refresh performed by Activate. But such refresh incurs larger overhead, because each bank has to be refreshed separately. The overhead is even larger for devices with more than 4 banks because only 4 activates can be performed in t_{FAW} time window.

The LPDDR (Low Power DDR) specification [JES10] has the same constraint as DDRx generations. The later LPDDR2 [JES11a] and LPDDR3 [JES12c] introduce per bank refresh command *REFpb*, which allows to refresh banks individually without requiring $9 \times t_{REFI}$ rule. But the new per bank refresh command is still subject to t_{FAW} constraint of 4 activates. In other words, the new commands do not introduce much flexibility, because it is equivalent to performing refresh by Activate. The only advantage is that the row counters are kept in SDRAM device.

Chapter 4

Real-Time Systems

This chapter provides some background on Real Time Systems (RTS). The chapter starts with the overview which contains the definition of RTS. In the second section we will introduce the model of task's computation and an example, which will be used to visualize some of the problems in RTS. Next we present the steps of the timing analysis. We discuss the requirements for a RTS platform and touch upon some of the inherent obstacles in achieving efficiency. Finally we make few notes about the T-CREST platform for real-time systems.

4.1 Overview

The RTS are systems in which computing the right output is not enough. The timing of the result is an integral part of the correctness. Often the exact timed behavior is not required, but rather guaranties of producing the output before the deadline.[1]

To implement and verify the system it is decomposed into individual periodic activities. For example, a control system would need to periodically sample

[1] when the exact time of action is needed, the response can be delayed until the right time comes if is computed too early

the values of the sensors, update its model of the environment and control the actuators. The computation needed to perform each activity is called a task. The task is invoked periodically and each such instance of the task must respond before the deadline. The tasks within RTS can be distinguished into classes by the implications of missing the deadlines [But11]:

Hard deadline. Missing a single deadline might cause significant damage and must be avoided. The damage could for example be economical, ecological and even loss of live. The worst case performance guaranties are ultimate requirement and the average case performance is irrelevant. The examples of such system could be heart pacemaker, traffic light control, nuclear power plant control etc.

Soft deadline. Missing a deadline is not critical, but late response leads to performance degradation of the system. The statistical timing guaranties can be used and average case performance might be of larger importance than in hard-RTS. The example of activity with Soft RT requirement is handling of the user interface, like displaying message on the screen or the keyboard input.

Firm deadline. Similar to Soft RT, some deadlines can be missed, but the late result has no value, i.e. can be discarded. While in the Soft RT it is important to provide the result even if it is a little bit late. In Firm RTS it might be beneficial to early drop the delayed activity in favor of meeting the future deadlines. The usual example of Firm RTS are multimedia applications where dropping the frame is less critical than processing it with a long delay.

[But11] expressed the differences between the RT and non-RT activities graphically (Figure 4.1). For non-RT task the usefulness of the computation result does not depend on time. For RT tasks the deadlines are usually specified in such way, that results produced any time before the deadline is equally good, and the difference is in outcome of the late result. For hard RT task, the value is minus infinity as it causes severe damage. The delayed result of firm RT task does not cause any damage, but has no value either, while it decreases over time for soft RT task. The main focus of further work is optimizations for hard RT tasks, though we try to have in mind the existence of other tasks in a system.

4.2 Modeling the Task

As mentioned previously, RTS applications are usually modeled as a set/graph of tasks. The task is single thread of computation with deadline requirement. We constraint ourselves to a simple task model from [BW01]:

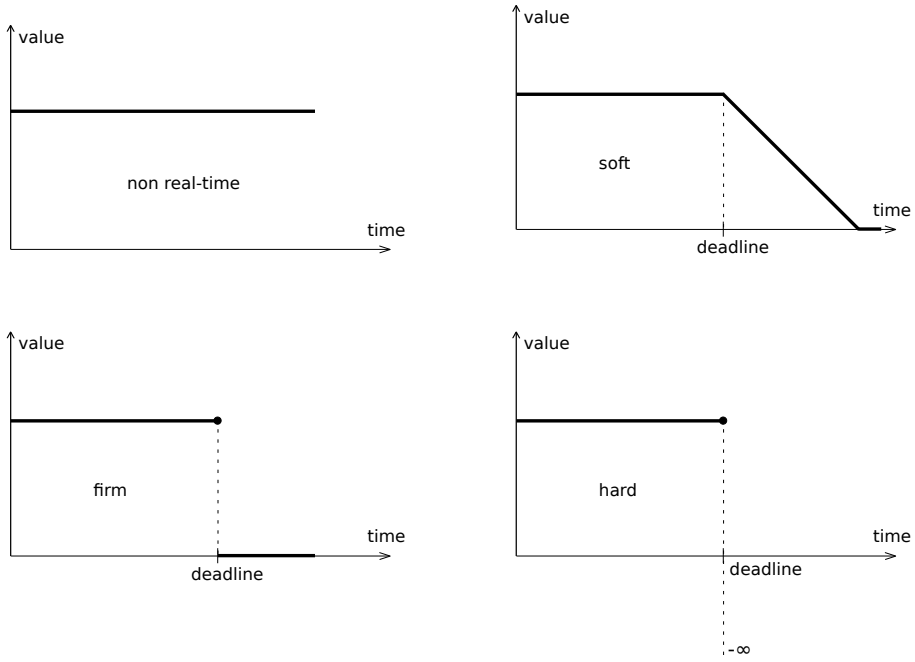

Figure 4.1: The usefulness of the computation result over time in different systems. The image from [But11]

- The tasks are periodic, with known periods, and the deadline is equal to the period.
- The tasks are completely independent. There are no critical sections; the inter-task communication is performed on task boundaries. That is, all the inputs are assumed to be ready at the time of task instance release and all the outputs are assumed to be produced at the end of the task's computation.
- All tasks have a fixed worst-case execution time (WCET). We will talk about it in following subsections.
- The overhead of context switch is assumed to be zero.

The elimination of these constraints is possible, but we want to limit our discussion to simple case. Actually, the zero cost context switch is possible if fine grained memory access scheduling is used and dedicated processor core per task is available as described in Chapter 4.5

Let us now look closer at the task level, and show one way of abstracting its behavior. The following listing shows hypothetical task described as C function.

The `scA()`, `scB()` etc, denote some straight line code. There is a conditional branching and a loop whose body execution is limited to 3 iterations:

```
int task(int in1, int in2) {
    scA();
    if (in1) {
        scB();
        for (int k=1; k <= in2 && k <= 3; k++) {
            scC();
        }
    } else {
        scD();
    }
    scE();
}
```

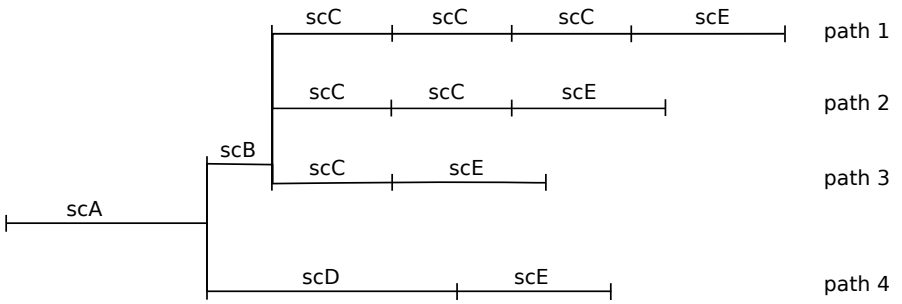

Figure 4.2: The possible distinct execution paths of the example program

The example demonstrates three sources of uncertainty which have to be dealt with to provide guaranty of meeting the deadlines: input, hardware state, analysis approximations. The branch taken and the number of loop iterations depends on the value of the input parameters `in1` and `in2`. This creates four possible execution scenarios, called *execution paths*. The paths are shown graphically in Figure 4.2. Each path would probably have different execution times, that is the execution time is *input dependent*. We now look at the `scE()` fragment. Even though it is executed at the end of every path, its execution time might actually vary. This is because the fragments executed before the `scE()` are different in each path, and might leave the *hardware* (for example the caches) in different state. To deal with it, the analysis is forced to either analyze each path individually or to use *approximation* of hardware state which is common to all the predecessors. The former option makes analysis difficult because there are potentially exponentially many paths. The latter option reduces the accuracy of the execution time estimate. Finally, because the task's code is executed periodically, the hardware state at the beginning of the tasks execution depends on

the state left by the previous instance and would also need to be approximated anyway.

4.3 Timing Correctness Verification

Proving the timing correctness of RTS traditionally consists of two steps. First, the task level analysis obtains upper bound on Worst Case Execution Time (WCET) for each task. Once the WCET bounds for each task are known, the schedulability analysis can be performed to find out if all the tasks in the system will meet their deadlines with the given task scheduling policy.

4.3.1 WCET Analysis

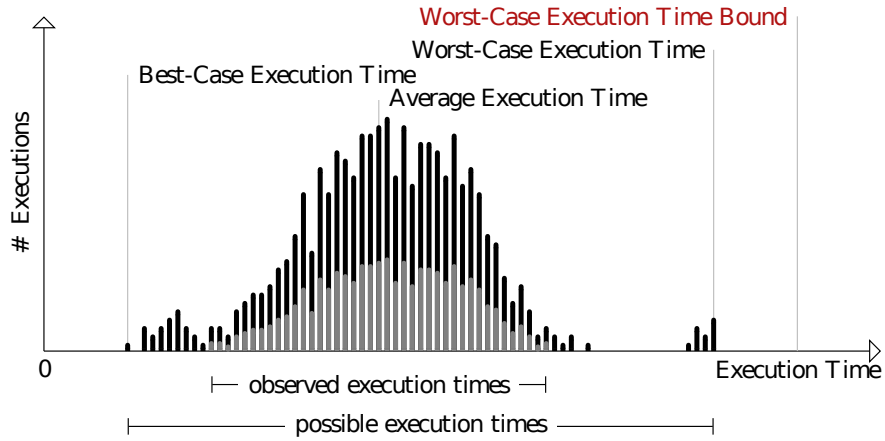

Figure 4.3: The histogram of possible execution times for some program. The lower histogram shows those executions which were observed during measurement which are only a subset of all possible executions (the dark histogram). The image from [WEE+08].

The WCET analysis must derive the upper bound guaranties on execution time for each task. [WEE+08] visualized the problem of WCET estimation graphically (Figure 4.3). The measurements are not safe, and the results of safe analysis are not tight. Ideally one would want to know exact WCET, but this is not possible through measurement because it is intractable to exhaustively ex-

plore exponential combination of inputs and hardware state of the computation. Therefore the execution time must be analyzed by using safe approximations, which leads to guaranteed but not tight WCET bound. [WEE+08] can be consulted for further information about the methods of the timing analysis.

We would like to point out, that it is in general unfeasible to know precise local time at the analysis point. This is because, as mentioned in the example in Section 4.2 the time to execute the predecessors of the current point can vary even on single execution path. Knowing the time could help the analysis to obtain more accurate bound on memory accesses for those arbitration schemes where the guaranties are different depending on time. For example in TDM arbitration, the response time depends on the current offset to the slot (see Figure 6.2) . The other example could be the schemes with replenishment period (DPQ described in Section 6.2.3 and PBS from Section 6.2.5). The time can be synchronized by introducing conditional delays in the code, but the waiting penalty should be smaller than the precision gain. The relative notion of time could be available in simple straight line of code, though this is also subject to accuracy of cache analysis at the point.

4.3.2 Schedulability Analysis

When multiple tasks are run on the system, they will usually interfere. They will compete for shared resources at the hardware level (processors, buses/interconnect and memories) and possibly perform some additional synchronization in software. The interference is usually variable because the instances of different tasks have different periods. Some scheduling policy is employed to limit this variability by constraining the execution order of the tasks. The schedulability analysis is then able to verify if all the deadlines will be met, assuming the WCET bounds for each task instance and known maximum interference allowed by scheduling policy.

The schedulability analysis for uni-processors is a mature research area and has well established results. The detailed coverage of analysis methods is out of scope of this work. It can be mentioned that the methods depend on employed scheduling technique.[2] For example, for the offline static execution order scheduling, the schedule is fixed upfront during the design time, so schedulability can be checked by simple simulation of the schedule. For static priority schedulers (for example with rate monotonic (RM) priority assignment), the schedulability can be checked by calculating the worst case interference of all the tasks (which happens when all tasks are released at the same time, so only

[2]The following examples assume simple task model, Section 4.2

this particular case need to be analyzed). For some dynamic scheduling policies, like earliest deadline first (EDF) the question can be efficiently answered by checking utilization bound, i.e. the sum of CPU time required by all the tasks. The scheduling for uniprocessor is nicely covered in [But11]. [CDKM02] in addition touches some issues of scheduling on multiprocessors and distributed computing.

The schedulability on multiprocessors is trickier, because some anomalies emerge and the results of uni-processor scheduling are in general no longer valid. Two main scheduling classes are used: *Global Scheduling* and *Partitioned scheduling*. Global scheduler uses a single queue for all the ready tasks and assigns them to available processors. This means that tasks might need to migrate to other processor after preemption. The main problem of global scheduling is that the single ready queue might not scale well and the overhead of migration might be very large. In partitioned scheduling each processor gets a fixed subset of the tasks which will be scheduled locally (i.e. migration is not allowed). Additionally, because the tasks are fixed to processor, the uni-processor schedulability results can be used. Unfortunately, finding the optimal partitioning of the tasks between processors is NP-Hard problem. Also the over-allocation of the processors is potentially larger than in case of global scheduling, because a ready task can not be migrated from the busy processor to the one which is idle. The multiprocessor schedulability results are especially sensitive to assumptions of the system model, like task deadline equal to the period, no critical sections etc. An in depth survey of the problems and results is covered in [DB11].

We would like to finish the overview of timing verification with the observation that on the multiprocessors there is some flexibility of handling the task interference caused by access to shared hardware like an external memory. In the uni-processors the cost of access to memory is traditionally incorporated into WCET, but it might be beneficial to analyze it at the level of schedulability on multiprocessing systems. We discussed in more details further on the example of handling the SDRAM refresh.

4.4 Platform Requirements

The ultimate requirement for the RTS platform[3] is to provide guaranties that tasks will meet their deadlines. That is, the derived WCET bound for each individual task will not be exceeded and that tasks will be properly scheduled in any combination of their interference. This leads to slightly different notion

[3] here platform is used in broad sense, combining the hardware and all the tools, including those performing timing analysis.

of performance, than understood by conventional computing. Next we look at the source of RT inefficiency and at proposals to deal with it. Finally we look at some additional requirements, which are not strictly necessary, but allow simplification in the analysis.

4.4.1 Performance

Intuitively, fast computer has short response time (low latency) and/or is able to complete a lot of work in short time (high throughput). In conventional computing both metrics are optimized for average case. The users prefer computer which seemed to be two times faster most of the time, even it would occasionally freeze for some time. Similarly, in throughput oriented computing it is preferred to complete 2 times more jobs per minute, even if some individual jobs would take 10 times longer to complete. Such reasoning is not applicable to RTS, because it is not acceptable to miss even a single hard deadline.

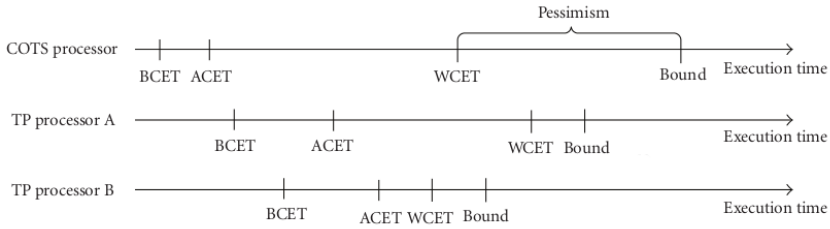

Figure 4.4: One conventional and two time-predictable architectures with different best-case, average-case and worst-case execution times and the WCET bounds. [Sch09c]

From the perspective of a single hard-RT task, the performance has two aspects: low WCET and, more importantly, its bound. The concept is visualized on Figure 4.4 reprinted from [Sch09c]. Even though conventional (COST – Component Of The Self) processor has the best average case performance, it is not a good platform for hard RTS as its guaranteed performance is worst. More interestingly, even though the exact WCET of conventional processor is lower than of the TP processor A, the latter is preferred because it has better guaranteed performance (lower bound on WCET).

Naturally, it is easier to guaranty that the tasks will meet the deadlines if it has lower WCET bound. But the RTS is rarely composed of a single task. The tasks would interfere at different levels causing their response time to be larger than the WCET when run in isolation. So, it is not only the latency of

individual task, but rather collective behavior is important. That is, it is valid and preferable to prolong the response of some task (provided that it is still within the deadline) if this helps to meet the collective deadlines of the system.

4.4.2 Timing Predictability

In previous section we have shown that in hard RTS only the guaranteed worst case performance is valuable. We now look at the source of execution time discrepancy. There are several related, but slightly different notions allowing to reason about the execution time variations [PKP09]. We look at some of them here having in mind the RT platform in general and RT memory controller specifically.

By definition of the adjective *predictable*, the future behavior of the predictable system can be estimated. Intuitively this is a very nice property for RTS, but it is too abstract. [Sch12] contains a survey of quantifiable definitions for timing predictability and comes to conclusion that they are still not very useful in practice. The overviewed definitions try to measure the predictability by quotients of BCET, WCET and/or their bounds. This is problematic because exact WCET are not known. So instead of looking for definition of predictability which could be used to measure our memory controller. We now consider predictability as qualitative property. We first consider the sources of unpredictability, and next look at some of the proposed predictability related architecture properties.

As it was demonstrated on the simple task example in Section 4.2 the variability in execution time has two main causes: *input dependence* and *unknown hardware state*. The initial input affects the control flow of the program.[4] The input determines which conditional branch is taken and how many iterations of the loops are executed. The unknown hardware state, for example content of the caches can be caused by the interference of the tasks, as well as the different paths of the same task. Both factors get amplified during the WCET analysis, because unknowns input/state have to be mapped to known safe approximations.

Reducing the uncertainties as well as allowing tighter approximations during analysis is an active research area. [TW04] look at the threats to predictability at different levels of the developed RTS: hardware architecture, software development for single task, task level interaction and distributed operation. [WGR+09] cover hardware architecture features in more details. [GRW11] contain survey of research efforts dealing with sources of uncertainties.

[4] This happens not only directly but more through intermediate values.

It is worth mentioning one common property of modern conventional architectures that is quite hostile for RTS, namely *Timing Anomaly*. Timing anomaly is a contra-intuitive situation where seemingly favorable change in conditions causes negative overall effect. [Gra69] showed an example of multiprocessor job-set which would experience the longer completion time if an extra processor is added or the execution time of each job is shortened. [RWT+06] review the timing anomalies in the context of RTS, provides the abstract definition and recognizes three classes of anomalies: Scheduling Timing Anomalies, Speculation Timing Anomalies and Cache Timing Anomalies. [CHO12] propose alternative definition and compare it against other definitions on a few examples. Some examples of scheduling anomalies can be found in [But11].

If architecture is prone to timing anomalies, it is not safe to assume that local worst-case will lead to global worst-case. This makes the WCET analysis more complex and/or more lose WCET bounds by preventing the state space simplifications [WEE+08]. The platform free from timing anomalies, not only leads to better analyzability, but can also simplify some of the hardware components as argued in the next section.

We now return out attention into two recently mentioned sources of execution time uncertainty, and look at the proposed strategies of their elimination. One proposed way of removing input uncertainty is single-path paradigm. The program is transformed into version where both alternatives of the branch are conditionally "executed".[5] Similarly the body of the loops are run for maximum number of iterations, with appropriate predicates to disable the effect of the unneeded iterations. The motivation and further pointers can be found in [PKP09]. The single-path programs however introduce a penalty of WCET increase caused by serialization of all the alternatives.[6] The input dependence can also be abolished through balancing all the execution paths by padding the faster paths with delays. The simple extensions to the architecture have been proposed to achieve this with little overhead [IE06]. This involves software visible hardware counters and the *deadline* instruction to stall the processor until the counter expiration.

To eliminate the effects of hardware uncertainty, the notion of *Timing Repeatability* has been proposed. Timing repeatable architecture ensures the deterministic execution time for given input, i.e. both the output and the time when they are provided are always the same. [EKL+09] argues that repeatability is more important than predictability. Clearly, it is easier to analyze programs executed on a repeatable platform, but there are more advantages. Repeatability allows making assertions about timing correctness of a program by testing.

[5] Instructions traverse through the pipeline, but the state update is predicated and will occur only for instructions from the correct branch

[6] Instructions from all branches have to be fetched and pass through the pipeline

Contrary, on non-repeatable hardware, observing correct timing during testing does not guaranty that the correct behavior would occur when the program is run another time (even with the same input).

In general, the timing repeatable architecture allows the variability to be present across executions with different inputs. But if input uncertainties are also eliminated as discussed previously the *Stable Execution Time* is achieved. That is there is no variability between the best-case and worst-case execution time.

4.4.3 Timing Composability and Temporal Isolation

In a complex RTS the *Timing Composability* property becomes important. That is the timing guaranties of a task should hold irrespective of the behavior of the other tasks. This allows easier integration and certification, because execution times of the tasks can be analyzed independently. There are some variations in the definition of composability. [Ake10] uses stronger definition where tasks are completely isolated, i.e. the task would have exactly same timing behavior irrespective of other tasks. We think that weaker property is still useful assuming that the system does not experience timing anomalies. That is the exact behavior of the task might be affected by the behavior of the other tasks, but timing guaranties would still hold even during changes of other tasks. Whereas for the stronger property, a name *Temporary Isolation* is more appropriate [BLL+11].

Though not strictly necessary for composability, the temporary isolation would have advantage of fault isolation. For example, if some requestor would violate its timing specification and start requesting more memory than promised, the rest of the system should function properly, if possible.

4.5 T-CREST Platform

This section will introduce the T-CREST platform and highlight its main features having the implications to memory controller design. The platform is still in research process, and might undergo changes. The main features are:

- A homogeneous many core system with at least 32 processor cores envisioned to allow a dedicated core per thread/task. The Patmos processor core is described in [SSP+11] and more technical details in [**?**].
- Each core has a number of dedicated on-chip memories: data cache, stack cache, method cache, scratchpad (SPM) and a inter-core communication

memory. The part of the stack cache might need to be stored into external memory on function calls if the new frame can not fit into he cache. The external memory access will also be required if the last in-cache frame is popped during the function return. The method cache is used as a replacement for the instruction cache and has an advantage of more precise state during the WCET analysis. On a miss, the method cache fetches the whole function body[7] from the external memory. Finally the scratchpad (SPM) allocation is guided at the compile-time by the analysis algorithm which generates the code for explicit management of SPM content by processor. The granularity of external memory accesses will depend on the SPM allocation algorithm.

- The cores can transfers the data between their local communication memories. The transfers are performed by statically scheduled network-on-chip. The schedule is configurable, allowing to setup the communication channels according the latency and bandwidth requirements.

Most of the time larger memory transfers would be performed, but the memory should also support the single word (byte) reads and writes for uncached memory access and if fine grained scratchpad allocation is used.

[7] Alternatively, for larger functions, known part of it.

Chapter 5

Single-Port Controller Implementation

This chapter explains the design and implementation of the single-port SDR (Single Data Rate) SDRAM controller which was made during the project. We start by looking at the general responsibility of memory controllers, and their usual organization. Next we briefly state our reasons for targeting the older generation of SDRAM and provide some analysis of the SDRAM interface. The rest of the chapter provides some information on design, implementation, integration and testing. The controller evaluation is presented in separate chapter (Section 7.1).

5.1 Responsibility and Organization of a Memory Controller

A controller is usually organized as the Single-port Controller which can be used by single requestor only and the separate interconnect and arbitration layer which allows the controller to be shared across multiple requestors (Figure 5.1). In this chapter we focus on single-port controller and discuss the multi-port design tradeoffs in the next chapter. It is convenient to maintain the

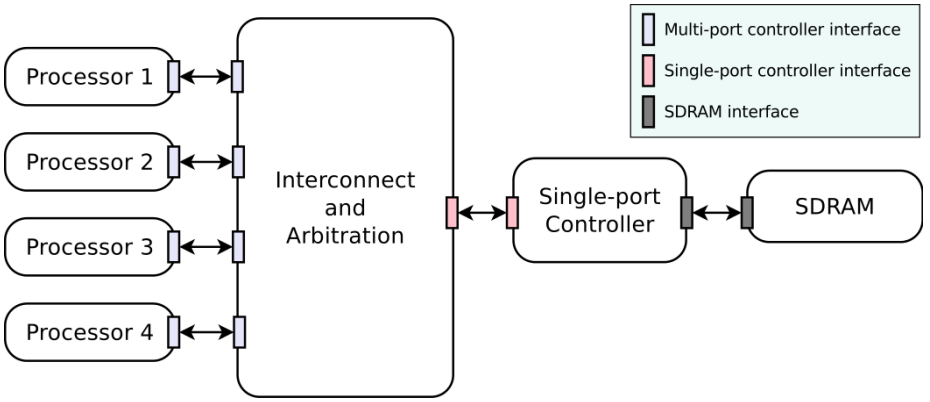

Figure 5.1: Controller Organization

same single-port controller interface for the multi-port controller to make the arbitration transparent, but it might be beneficial to optimize the multi-port interface. The single-port controller translates requestors' (processors' or DMA controllers') memory accesses into legal sequences of SDRAM commands. The linear memory addresses provided to the controller are translated into tuple of rank/bank/row/column addresses. The controller must also keep track of the state of the SDRAM banks and ensure that all the SDRAM timing constraints are obeyed. Usually, the controller is also responsible for issuing SDRAM refresh operations.

5.2 Motivation for Choosing the SDR Generation

Even though the SDR is currently quite old generation of SDRAM it is still used on some embedded platforms [BM11]. The interfacing is simple, allowing implementation not needing too much focus on the peculiarities of the signal integrity and write/read leveling needed in latest generations of SDRAM. Finally, the last reason for targeting SDR SDRAM generation was the use of the Terasic DE2-70 FPGA board, which is quite widespread in education. It is also currently used for development of the Patmos processor for the T-CREST project.

5.3 Analysis

5.3.1 Timing Parameters of SDRAM on DE2-70 Board

The Terasic DE2-70 board used during the project features two IS42S16160B-7TLI SDRAM chips [IS407]. They are organized as 16-bit words in 4 banks of 8192 rows by 512 columns. The chips are speed grade 7, so according the specification they can run up to 143 MHz (with CL=3) or 100 MHz with CL=2. The general timing parameters of the SDRAM were explained in the `Timing parameters` section of the 3 chapter. The values in the Table 5.1 lists SDRAM parameters for the IS42S16160B chip used during project. While the Table 5.2 list the parameters relevant for signal integrity.

Table 5.1: Relevant SDRAM(SDR) timing parameters

		7 ns clk (cycles)	8 ns clk (cycles)	10 ns clk (cycles)	Min (ns)	Max (ns)
	Clock Frequency (MHz)	143	125	100		
t_{CAC}	CAS Latency	3	3	2		
t_{RRD}	Row to Row Delay	2	2	2	14	
t_{RCD}	Row to Column Delay	3	3	2	20	
t_{RAS}	Row Access Strobe	7	6	5	45	120K
t_{RC}	Row Cycle	10	9	7	67.5	
t_{RP}	Row Precharge	3	3	2	20	
t_{CCD}	Column Command Delay Time	1	1	1		
t_{DPL}	Input Data to Precharge	2	2	2	14	
t_{DAL}	Input Data to Activate	5	5(4)	4	35	
t_{RBD}	Burst Stop to High Impedance	t_{CAC}	t_{CAC}	t_{CAC}		
t_{WBD}	Burst Stop to Input in Invalid	0	0	0		
t_{PQL}	Last Output to Auto-Precharge	1-t_{CAC}	1-t_{CAC}	1-t_{CAC}		
t_{QMD}	DQM to Output	2	2	2		
t_{DMD}	DQM to Input	0	0	0		
t_{MRD}	Mode Register Program Time	3(2)	2	2	15	
t_{REF}	Refresh Cycle (8192 rows)					64M

5.3.2 Separation between Transactions

We now find the minimum number of cycles needed between two operations can be issued. We first look at the operations with random address, and later at the operations with known banks, which will be used in next section to find some efficient command sequences for interleaved transactions.

Table 5.2: Timing requirements for valid signaling in ns

Symbol	Parameter	Min	Max
t_{AC2}	Access Time from CLK (CL=2)		6.5
t_{AC3}	Access Time from CLK (CL=3)		5.4
t_{OH2}	Output Data Hold Time (CL=2)	2.7	
t_{OH3}	Output Data Hold Time (CL=3)	3	
t_{HZ}	CLK to High Impedance Time	2.7	5.4
t_{LZ}	CLK to Low Impedance Time	0	0
t_{SU}	Input Setup Time(1)	2	
t_H	Input Hold Time(1)	1	

For random addresses the worst case is when consecutive accesses happen to the different rows of the same bank. Because we are interested in optimizing worst case performance, we use the closed page policy, because it assures smaller worst case latency. The read transaction would require: row activation, CAS latency, burst transfer cycles and precharge. In this case the precharge can be overlapped with last few data transfers cycles, the t_{PQL} term in future equation accounts for this. The +1 is added, because t_{PQL} contains the first cycle of the *Precharge* (see Section 3.4.2) The *max* is used to satisfy the t_{RAS} for smaller burst lengths (BL), and the whole sum must always be at least t_{RC}:[1]

$$Cycles_{RandomRead} = max(t_{RC}, max(t_{RCD}+t_{CAC}+BL-(t_{PQL}+1), t_{RAS})+t_{RP})$$

The Write transaction would require: Precharge, Activate, Burst transfer, Write recovery cycles. The write recovery cycles are $t_{DPL} - 1$, because t_{DPL} contains the cycle of the Precharge operation (see Section 3.4.2). Again, the whole sum must be at least t_{RC}.

$$Cycles_{RandomWrite} = max(t_{RC}, max(t_{RCD} + BL + (t_{DPL} - 1), t_{RAS}) + t_{RP})$$

The separation of operations to different banks is affected by the length of the burst and two timing parameters: the separation between Activates (t_{RRD}) and the Read/Write commands (t_{CCD}). For SDR the smallest meaningful burst length for interleaving is 2, because each operation requires at least two command bus cycles (Activate and Read/Write). Because the t_{CCD} and t_{RRD} are usually not greater than 2 clock cycles, the separation becomes BL. This means that two consecutive memory operations of the same kind (Read or Write), can result in uninterrupted transfer on the data bus. It is only left to look at separation between operations of different direction. The Read after Write is constrained by t_{RTW}, but for SDR memories this is always one clock cycle (that is commands can be issued in consecutive cycles). Because the write data transfer

[1] All the timing parameters used here are rounded to full clock clycles.

starts during the same cycle as write command, the minimal separation between the commands is BL as in previous case, but this creates t_{CAC} idle cycles on the data bus. That is the Read command is issued in the next cycle after last data written, but read data comes only few cycles later. The Write after Read needs an extra $t_{CAC} + 1$ cycles between command separation. The t_{CAC} are needed to let the read data to finish, and 1 extra idle cycle, which is needed to allow the tristate buffers of the SDRAM to enter become high impedance before the controller starts driving the data onto the bus. In principle this gap could potentially be optimized away, because SDRAM does not require the full cycle to enter high-impedance (Section 5.3.1), but this would require ensuring that the FPGA starts driving the data bus later.

Table 5.3: Minimal separation between SDRAM(SDR) transactions. The subscript$_{n-1}$ denotes previous operation, and b_{n-1} the bank it used. BL denotes Burst Length, all other timing parameters are represented as number of full cycles.

	$\text{Read}_{n-1}(b_{n-1})$	$\text{Write}_{n-1}(b_{n-1})$
$\text{Read}_n(b_n!=b_{n-1})$	BL	BL
$\text{Write}_n(b_n!=b_{n-1})$	$\text{BL}+t_{\text{CAC}}+1$	BL
$\text{Read}_n(b_n=b_{n-1})$	$\max(t_{\text{RC}},t_{\text{RCD}}+t_{\text{CAC}}+$	$\max(t_{\text{RC}},t_{\text{RCD}}+$
$\text{Write}_n(b_n=b_{n-1})$	$+(\text{BL}-(t_{\text{PQL}}+1)+t_{\text{RP}}))$	$+\text{BL}+(t_{\text{DPL}}-1)+t_{\text{RP}})$

Table 5.4: The maximum percentage of data transfer cycles for random operations. The bottom part of the table contains numbers for interleaved transactions from next section. The timing parameters are for 100 MHz operation of SDR memory on DE2-70 board. With the higher frequencies, more cycles will be needed and efficiency will be lower.

BL	$\text{Cycles}_{\text{Read}}$	$\text{Cycles}_{\text{Write}}$	Bw_{Read}	Bw_{Write}
1	7	7	14.28%	14.28%
2	7	7	28.57%	28.57%
4	8	9	50%	44.44%
8	12	13	66.66%	61.53%
2*2	8	8	50%	50%
2*4	8+3	8+3	72.72%	72.72%
2*4+2*4	$\frac{8+3+8}{2}$	$\frac{8+3+8}{2}$	84.21%	84.21%

The separation requirements are summarized in Table 5.3. The worst case fraction of maximum bandwidth for different burst lengths is presented in Table

5.4. The top part of the table lists the numbers for the simple transactions. The data is calculated according the presented formulas. For slotted schemes like TDM (Section 6.2.2), the cycles of the longer write transaction will have to be reserved, so both direction would have the same percentage from Bw_{Write} column. The estimates in the table do not account for cycles wasted for refresh and will be even lower. The bottom of the table contains 3 examples of interleaved transactions, which we describe in the next section.

5.3.3 Interleaved Transactions

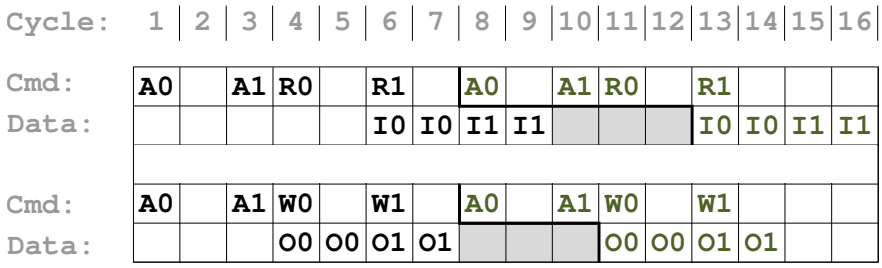

Figure 5.2: Four word operations interleaved over two banks. The top part shows back to back read operations, while the bottom part shows writes. The change of the read/write direction does not incur additional separation, because in both cases the new activate can be issued in cycle 8. There will also be required separation on the data bus, because the read finishes transfer in cycle 9, while the write starts it in cycle 11.

As it was shown in the end of Section 3.4.2, splitting longer access into several smaller transactions interleaved across several banks has the advantage of shorter cycle between unrelated operations. The same timing parameters as in the previous section are assumed. The minimal separations for both different and the same banks from the Table 5.3 can be used to find the command sequences for efficient interleaved transactions. We present three manually composed sequences for operations of 4 and 8 words. The interleaving for longer operations can be created in the same fashion. Some conventions are used in the following figures. $A < i >, R < i >$ and $W < i >$ denote the activate, read and write command to bank i, while $I < i >$ and $O < i >$ show the data input and output from the bank i. The commands of two consecutive transactions are color coded, because they interleave in the third example (Figure 5.4).

Figure 5.2 shows operations of 4 words with separation between operations of

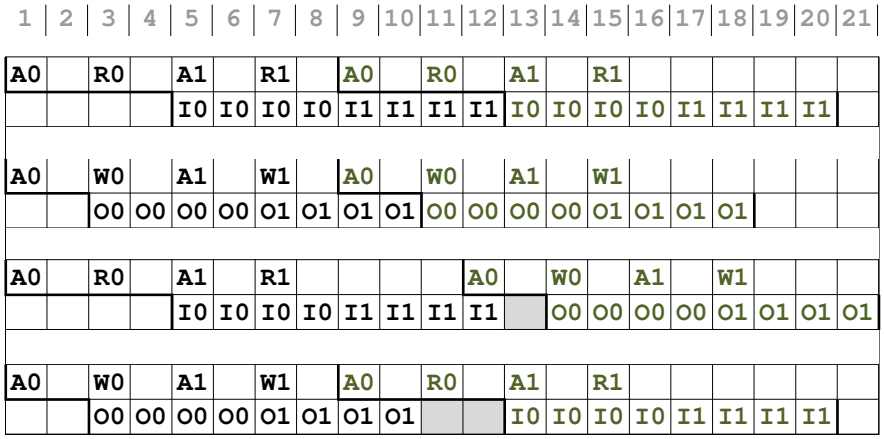

Figure 5.3: Eight word operations interleaved over two banks. First two rows show back to back reads and writes, the next rows shows that happens on the switch.

8 cycles. The Read/Write commands are used with auto-precharge to save the command bus from explicit precharge. The advantage of this approach over the non-interleaved transaction from the previous section, is 1 cycle saved for Write operation, making the slot shorter. Though the bandwidth gain is not impressive, the overall WCMAT would be reduced from $N * 9$ to $N * 8$ cycles, which might be noticeable for larger number of requesters. To get better bandwidth, the operation size has to be increased.

Figure 5.3 shows example of 8 word operation which can be repeated every 8 cycles. It can also be interleaved over the four banks almost exactly the same way (Figure 5.4), but it results in an additional latency of one clock cycle. The sequences are optimal with respect to efficiency, because the data bus can potentially be used in each cycle, except for Read/Write switches, resulting in 3 unused cycles for Read-Write-Read sequence. The length of the slot can be created to accommodate the longest separation between Activates of subsequent operations. In this example it would be 11 cycles needed for Read if it is followed by Write, and would allow to guaranty $\frac{8}{11} = 72.72\%$ of bandwidth (minus some used by refresh). But the extra 3 cycles would only be needed in every second slot (for alternating operations), this would require extra logic in the controller and possibly the arbitration/interconnect, but bandwidth guaranty could be raised to $\frac{8+8}{8+3+8} = 84.21\%$. The WCMAT would also be reduced from $N * 11$ to $N * 9.5$ cycles.

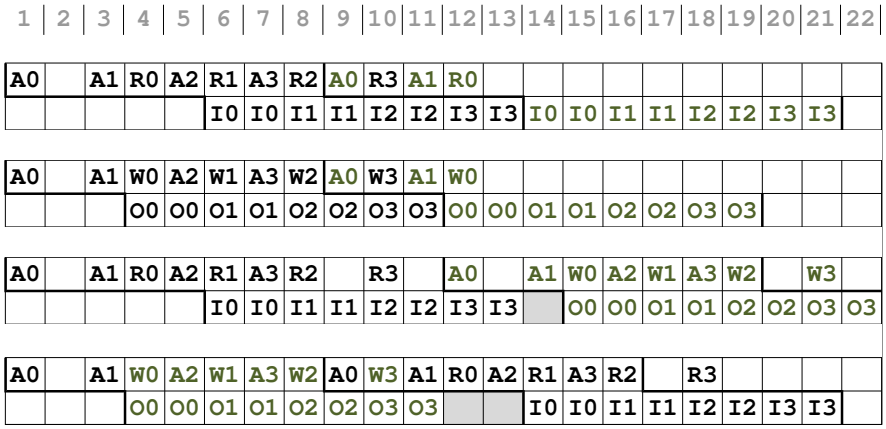

Figure 5.4: Eight word operations interleaved over four banks.

5.3.4 Performing Refresh

The issues of performing DRAM refresh in analyzable way were presented in Section 6.3.3. The SDR SDRAM memory targeted by the controller supports all the available options.

The dedicated Auto-Precharge command needs the same amount of time as regular pair of Activate and Precharge (t_{RC}), so it does not impact the slot size for simple transactions. While for interleaved transactions the refresh should be performed manually by Activates interleaved in a same way. That is the refresh of the first bank can be overlapped with the previous transfer from the last bank etc. In case of interleaved burst of length two (for example Figure 5.4), the Precharge commands won't fit in to the transaction sequence. The write operation sequence with disabled data mask signal can be used to perform refresh in such case.

5.3.5 SDRAM Initialization

The controller has to perform the SDRAM initialization sequence:

1. Wait for power-up and CLK stable.
2. No operation for 200us.
3. Precharge all banks.

4. The sequence of 8 Auto-Refreshes cycles, with the usual timing requirements.
5. Configure the device by programming the Mode Registers.

In case of SDR SDRAM, the initialization sequence is simple and requires some additional states in the controller's state machine. Though some controllers delegate this responsibility to software.

5.4 Design

Even though interleaved transaction have potential to utilize the data bus better, to start with, we have decided to build controller using simple, non-interleaved transactions. The simple transaction controller issues the same sequence of commands for different configurations. This way it can be made configurable, where the target frequency and burst length are specified as generic parameters and all the waiting cycles are calculated automatically by the synthesis software.

Controller can later be extended to use the bank interleaving. This would potentially require few different controllers as the command sequences for each configuration could be different. As it can be seen from presented examples of interleaved operations, the command patterns can be very regular, so the implementation would be simple.

The controller needs to perform predefined sequence of actions for each operation, and this can be controlled by a state machine. First the initialization sequence has to be performed, after which the controller waits for memory access requests. The mealy type state machine is chosen to save one clock cycle of latency on the requester's interface. The mealy type does not introduce any bad effects on the SDRAM interface, because the output signals are registered in IO cells, to have good clock-to-output timing.

It is decide to use non-proprietary interface standard to allow the reuse of the controller. The Open Core Protocol (OCP) interface is chosen. The out of band signals defined in the OCP protocol are used for manual triggering of refresh (if the option of automatic refresh is disabled).

5.5 Implementation

The implementation has following features:

- Simple RTL state-machine in one entity. The result is comprehensible and easily maintainable code which is synthesized into reasonable implementation by standard vendor tool chain (see Section 7.1)
- The code is meant to be portable so vendor specific components/design patterns are tried to be avoided. The only vendor specific component is the PLL/DCM or similar which is necessary for higher operation frequencies, but it is instantiated externally to the controller. Also the use of IO-Block registers has to be specified in vendor specific way. For Altera the signal attributes are used, but .qsf can also be used. For Xilinx, this can be specified on per signal basis in constraint file.
- All the possible parameters are configurable through generics:
  - The signal widths for both requester's interface (address/data) and SDRAM interface (chip-selects/banks/address);
  - Address mapping from linear address of the requester to rank,bank,row and column;
  - Burst size (only the elementary sizes directly supported by the SDRAM specification);
  - Frequency, access latency, refresh period and other timing parameters from chips specification. Timing parameters are specified using the VHDL time constants and are translated into required number of clock cycles according the specified clock period automatically.
- Uses registers in IO-Blocks for better setup times and clock to output delay.
- Uses wait states and binary counters to insure timing between SDRAM bus commands. These counters were sufficient for the needed wait ranges.

Timing analysis reported maximum frequency is above 200 MHz (the highest speed of modern SDR chip), while memory used on DE2-70 can only support 143 MHz. The critical path goes through shared 15-bit counter used to wait for $200\mu s$ during the SDRAM initialization and for measuring the period between refresh need to be invoked. The path can be optimized in few ways if needed. First of all, both counting and a check for the counter expiration is performed in the same cycle, this can be split into two cycles by introducing a register for counter expiration signal. The carry chain can also be broken the same way, because the exact value of the counter is not needed, rather than its expiration condition.

The 3 ns skew is introduced between the SDRAM clock and the clock used for the controllers state-machine, to adjust the clock edge with the data for the same setup/hold slack for both read and write operations as described in [Alt09].

5.6 Integration

This section describes the controllers integration with two processors for RTS.

5.6.1 Integration with the Patmos Processor

This section describes the controllers integration with the Patmos processor (Section 4.5). At the time of writing the Patmos was still being developed. The support for caches was not yet finished and the processors pipeline used simplified access to local memories without stalling. The integration was performed through simple I/O controlled DMA (Direct Memory Access) like device. The device can be asked to perform external memory transfers with its local buffer. The single cycle access is provided to the buffer which does not require stalling the processor. Instead the processor polls the device status to find out if the memory transfer has been completed. Some more details on the device can be found in Section D.3.

5.6.2 Integration with the JOP Processor

JOP [Sch09a] is time-predictable processor for RTS implemented in Java. Even though the JOP wasn't the main target of this work, the initial integration was performed, because it is envisioned to also use the controller in the JOP based systems.

The JOP processor accesses the memory and I/O devices through SimpCon [Sch09b] interface. The interface is optimized for processor's pipeline. The processors drives the output signals for one clock cycle and waits until slave completes the transaction. The interface allows the slave to perform the early completion acknowledgements, by providing the master a hint on how many wait cycles are needed before the data is ready. 2-bit `rdy_cnt` signal is used for this purpose, where the value 3 has a special meaning of unknown number of cycles. The slave is required to keep the values of the input data and the acknowledgement after the transaction is complete.

A small VHDL entity was created to adapt the controller to SimpCon interface. Non-optimized adaptation is used, without early acknowledgement and controller configured to burst length of 1. When the transaction is issued the adapter sets the SimpCon `rdy_cnt` signal register to 3 (busy with unknown completion time). The command, address and wr_{data} signals are also registered

to keep them stable as required by the controller. When the controller acknowledges the data, the `rdy_cnt` register is reset to 0 (ready, i.e. zero wait cycles left), and register the input data for the read operation.

The first deficiency can easily be solved, because controllers keeps track of when the data will be available, and this information can be used for `rdy_cnt`. Solving the second deficiency requires modification of the controller and/or extensions to SimpCon interface. The SimpCon supports pipelined transactions, and could be used for SDRAM if some signals are added to denote when the transaction corresponding to the same burst are finished. The controller could than use this information to start the bank precharge. In addition the controller would need to be modified to support pipelined transactions.

5.7 Testing

Two testing methods are employed: VHDL testbench for controller simulation and a test programs for checking the controller operation in the on FPGA system.

5.7.1 VHDL Testbench

The testbench (TB) for RTL level simulation of the controller was used to test its behavior in isolation. The TB was also useful for locating the source of flaws, because the complete observability of the controller's state is available during the simulation.

The initial TB source was reused from the one created for the patmos integration with TU/e memory controller (Appendix D). The approach has an advantage that the controller is tested at the processors interface, but complicated the TB unnecessary. The TB created from scratch operating at controllers interface could be a cleaner alternative.

The TB performs writes and reads with different addressees and check that the read data matches the one written to that location. The mismatched entries are reported. There are also flags controlling the verbosity level, to control the reporting of the transactions on both the controller and processor interfaces. The reporting allows to run the TB in batch mode, for simple check of correctness without the need to examine the signal waves.

The TB does not try to test all the configurations supported by the controller, because there are many. Instead, one configuration is tested, which is selected by specifying the configuration constants. The Refresh period was configured to a small value to check the Refresh logic interference with the regular transactions. While the controller's conformance with the SDRAM timing constraints is verified by the SDRAM simulation model. The initial use of the model introduced some simulation mismatch. The reason was the clock gating performed inside the model, which resulted in delta cycles discrepancy between the controllers clock and the internal clock of the SDRAM model. The problem was resolved by introducing a small propagation delay for signals on SDRAM interface. This way the data sampled by both clocks was corresponding to the same logical clock cycle, and behavior was equivalent to synthesized implementation.

5.7.2 In System Tests

The FPGA synthesized version is tested in test programs executed by Patmos processor (Section 4.5). This way controller is tested together with the integration logic. A simple controllers test was also performed on the JOP processor.

5.7.2.1 Patmos Test Programs

The synthesized version of the controller was tested on the FPGA configured with a system composed of Patmos processor, memory controller accessible through I/O controlled DMA-like device (Section 5.6.1) and an UART for communication with the PC over RS-323 cable. The test programs were written in C programming language. At the time of writing only C programs with limited features could be executed successfully by the available infrastructure. In required some trial and error effort and examination of the compiler generated intermediate code before two simple test programs were made.

The first program (`test_sdram.c` Appendix A) uses only small part of the memory. The test consists of few steps. First the memory mapping of the I/O device is checked. Next some strings are written to the memory, read back for comparison and are output to serial terminal for examination. The second program (`test_sdram_full.c`) tests the whole addressable memory range. Distinct values are written into each memory location. Before the read-out and comparison is performed, the program waits for any input from the serial terminal. This is used to check if the SDRAM is refreshed to preserve the correct values. Program report the error when it occurs as well as the number of all errors discovered at the end.

5.7.2.2 JOP Hello World Test

The simple test was used to check the JOP integration (Section 5.6.2). The JOP system with SDRAM as the only external memory was configured on FPGA. Than the "Hello World" program was transmitted to the FPGA over the RS-323 cable. The bootloader received the program into the external memory and executed it from there.

Chapter 6

Multi-Port Controller Design

This chapter discusses the real-time memory controller for multiprocessors contains a discussion of the problem domain and an overview of the possible solutions and their tradeoffs. We start by looking at the controllers efficiency requirements and extend the task model from Section 4.2 which would allow to compare the efficiency of the memory controller arbitration schemes. Next we consider different arbitration schemes and see their possibilities and weaknesses. Then we look at the tradeoffs stemming from SDRAM interface properties.

6.1 Controller's Efficiency

It has been suggested in the Chapter 4 that the resource over-allocation is unavoidable in hard-RTS. This is because additional resources must be reserved to cover all the uncertainties in worst case. The efficient memory controller minimizes the over-allocation by limiting the uncertainties. Unfortunately it is not only intractable to completely eliminate over-allocation, but it is also impossible to create a memory controller which will be optimal in general. The platform can only be optimized for particular class of use cases. We will try to point some abstract properties of the applications that might affect the efficiency of particular controller organization. Next we will extend the task model to demonstrate some examples of performance variations for each arbitration scheme.

6.1.1 Modeling Memory Requirements of a Task

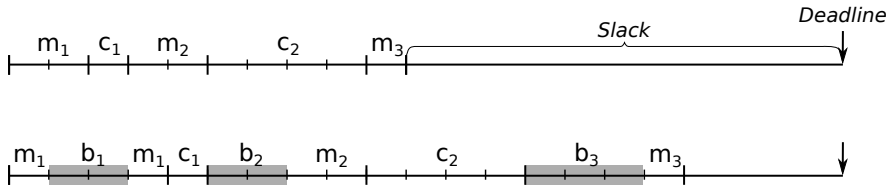

Figure 6.1: The single instance of the task viewed as a sequence of computations and memory accesses. Top: the task execution if the memory would always be granted. Bottom: the task got blocked for some time on each memory access.

In Section 4.2 we presented the view of a task consisting of many possible execution paths. From the perspective of the controller it is useful to have a more detailed view of a single path. The execution time can be divided into computation and memory access parts. Figure 6.1 shows a single instance of the task from its release to the deadline. The top part shows the optimistic execution where every memory access is performed immediately by the controller. One can also see that there is some slack between the completion of the task and its deadline. This slack is a "safety buffer" which must be reserved to cope with blocking uncertainties. The bottom part of the figure shows more realistic execution where the task experienced blocking on each memory request, and hence its completion (execution time) was delayed. To meet the deadline the sum of all the blocking should be within the available slack.

Because we are interested in the memory controller's point of view, the notion of the computation should be interpreted as any activity not requiring access to external memory. In addition to regular computation with processor registers this could include on-chip memory access (cache-hit, scratchpad read/write) or just a busy waiting. The important part is that the bounds of the duration of each computation fragment can be calculated by WCET analysis. If the predictable architecture is used the WCET of the single task instance is the sum of WCET of computation fragments and WC memory access times. Here, memory access time is a sum of fixed time needed for a transfer and a varying blocking caused by waiting for access to shared resource.

The example leads to two metrics of the controller's efficiency. From the perspective of the single task the controller should minimize the memory access time, i.e. have low bounds on blocking times. But from a global perspective of the system the memory bandwidth over-allocation should be minimized (obvi-

ously with the condition that deadlines are still met, i.e. the sum of blocking time is less than the execution time slack). This is because on a multiprocessor computations can happen in parallel, while memory access parts are serialized by the single memory. That is with sufficient number of cores the performance of the system will be limited by the available bandwidth. Both metrics are important for efficiency of the RTS, and we will try to estimate them when comparing different memory access scheduling schemes.

6.1.2 Notation

We would like to present the notation used for calculation of processor blocking time and memory bandwidth over-allocation in following sections. We will not try to rigorously calculate precise quantities, instead we make few simplifications which will allow us to make simple and intuitive estimates. We assume the slotted access to the memory, where fixed number of words can be read(written) from(to) the memory. We neglect the time needed to propagate the memory request from the processor to the memory controller, and to transmit the data back. We also assume the computation time to be represented in integer number of slots. The simplifications allow to have common unit of measurement for computation and blocking times as well as for memory demand and over-allocation:

- M: the total number of memory transfer slots needed by the task in some interval of interest (for example the period of a single task or hyper-period of all the tasks). That is, the M is cumulative memory bandwidth in this interval. If the precise number is not known, the worst case upper bound is assumed.
- $\mathrm{M}_{\mathrm{waste}}$: the memory bandwidth over-allocation, i.e., a number of slots allocated for the task that are never used. The slots that are allocated for worst case demand, but are not used for some inputs are not considered wasted here, because they are necessary for worst case behavior.
- C: the total on chip computation time excluding the external memory operations.
- B: the total time blocked on memory access.
- T: task's period time, i.e. separation between two instances of the task.
- D: task's deadline time, relative to the task release time. In simple task model this is equal to T.
- N: number of requesters.
- k, n: number of task allocated and the total number of slots in TDM allocation period.

6.2 Memory Access Scheduling

We have made some reasoning about the tasks demand of the memory, now we will overview what options are available for sharing the memory among multiple tasks. We introduce the different schemes by presenting some general scheduler properties. And later we go through the mentioned schemes and look at how each can be used in RTS.

6.2.1 General Scheduling Classification

Scheduling is about organizing in time the access to some shared resource. Sometimes the term access arbitration is also used for this purpose, though the arbitration is more a local low-level decision of who will use the resource now. While the scheduling is a higher level idea of how the resource should be shared. [Ern04] uses following classification of the schedulers :

Static execution order scheduling is compile-time precomputed schedule. Each computation, memory access or communication gets assigned fixed time interval. The global state of the system is modeled during scheduling and the resource contention is avoided. This approach is similar to for example train schedules, where each train is assigned fixed location for each time. The periodic tasks are handled by creating the schedule for a hyper-period (least common multiple of all the periods), which is then repeated all over again. The created schedule is fully deterministic, so interference uncertainty is fully eliminated, but creation of optimal schedules is an NP-hard problem, and it is thus only possible for small systems with small hyper-period. If it is possible, then because the dedicated processors are available per task, only the memory access needs to be scheduled and it can be implemented as a TDM based schedule with arbitrary allocation described in Section 6.2.2, though the period would probably be significantly larger.

Time-driven scheduling divides the time into slots and assigns them to requesters independent of the global state of the system. Two subclasses can further be distinguished:

Fixed time slot assignment to requestors. It is decided upfront at which relative time the requester will be allowed to use the resource. The difference from the static execution order scheduling is that the full system state is not modeled, and it is only the slot assignment that is deterministic. This covers different variations of time division multiplexing (TDM). Each requestor periodically gets an exclusive access to the resource. The main advantage of TDM is that re-

questers are isolated, i.e. the same service is provided independent of the activity of other sequesters. The drawback is that because time slots are exclusive, the idle slot can not be used by other requestor.

Dynamic time slot assignment does not fix the assignment upfront. Instead it will depend on the runtime behavior of requesters. For example round-robin (RR) grants access to next interested requestor in cycling order. This way requestor can not predict its slot location, as slots can be shrinked or omitted depending on access pattern of other requesters.

Priority driven scheduling distinguishes requestors according to importance, i.e. higher priority requestors are serviced before the other. Again, there are two subclasses:

Static priority assignment associates fixed priorities to requestors, which do not change during the whole system use.

Dynamic priority assignment will change the requestors' priorities according some rules.

We further mention two other properties often used when describing schedulers:

work-conserving scheduler will always grant access to the shared resource if at least one requestor is interested. Or analogously, the requestor can only be blocked if some other requestor is using the resource. It might seem counteractive at first, but a non-work-conserving scheduler can provide better latency guaranties in some context (Section 6.2.4).

preemptive scheduler can stop currently serviced requestor in favor of later arrived more urgent request. Early generations of SDRAM actually support interrupting the currently active burst transfer, but this does not provide much benefit if small transfers are used. The effect of preemption for larger block transfers can be achieved by always performing it as a sequence of smaller transfers and doing fine grained arbitration.

6.2.2 TDM: Time Division Multiplexing

The main advantages and disadvantages of TDM stem from its static nature. The static, up-front fixed knowledge could allow the analysis to make tighter bounds and allow the hardware to be optimized. On the other hand this does not allow adapting to runtime conditions.

Because of its static allocation the TDM is not work conserving, so precious memory bandwidth is wasted even when there are requesters waiting for it. However compared to work conserving Round Robin discussed in next section, the TDM allows to predictably overlap the computation with the memory access

waiting, so if the computation is shorter than the separation between allocated slots, the waste is eliminated and the lower latency is provided. For system with many requesters, the period between the allocated slots will be large, usually there might be enough time to complete the computation. Finally, it is simple to incorporate best effort requestors in the idle slots. So if the system contains some non-RT tasks they can consume the potentially wasted bandwidth. Moreover, as we show later, the amount of bandwidth over-allocated by hard-RT task (i.e. left to other) can be calculated, so soft-RT can also be supported to some extent.

The TDM is easy to analyze. The requesters are isolated, so timing repeatability and composability can be maintained. The isolation allows making the whole analysis at the intra-task level (i.e. WCET analysis) without the loss of precision. Moreover the worst case memory access time (WCMAT) can be precise if the time offset in the allocation table is known at the WCET analysis point. For allocations with the regular period the memory request has a side effect of synchronizing to the slot table. For example for multiple read requests with known length of the intermediate computation, only the first request would need to assume the WCMAT, while for subsequent requests the exact latency is known.

The memory bandwidth over-allocation for one tasks instance is approximately equal to:

$$M_{waste} = T\frac{k}{n} - M \tag{6.1}$$

Here the $T\frac{k}{n}$ and M are respectively the total allocated and the sum of used memory bandwidth during the one task's activation period T (T is equal to the task's deadline in the simple task model). k/n represents the fraction of allocated bandwidth, where n is number of slots in the slot allocation period, and k is number of slots allocated to the task. The CPU over-allocation, i.e. the time wasted while being blocked on memory requests, depends on the distribution of the memory requests. For the regular allocations with equal separation between slots the CPU blocking in worst case is equal to

$$B_{max} = M\frac{n}{k} \tag{6.2}$$

Here $\frac{n}{k}$ is worst case latency for one request and should be updated appropriately for the arbitrary slot allocation. The worst case occurs if the memory is requested in one big chunk, or all the requests happens to arrive one cycle too late and need to wait a whole period until the next slot. For the exact estimate the computation cycles that are overlapped with memory blocking must be subtracted from the worst case blocking (i.e. in worst case no computation is overlapped).

Few interesting observation can follow from the interdependence of the equations

6.1 and 6.2. The tasks period is usually specified by the application domain, and is fixed. So the only way to reduce the bandwidth waste is to reduce the number of allocated slots k (eq.6.1). But this has an effect of increasing the B_{max} and can only be performed until the blocking is smaller than the slack separating the WCET and the deadline. So it is essential for the WCET analysis to find as much guaranteed overlapping as possible, because the pessimism of eq.6.2 leads to unacceptably high over-allocation:

$$M_{waste}^{pessim.anal.} \geq (C + M)\frac{k}{n} \tag{6.3}$$

Side note: *Derivation of eq.6.3.* Tasks computation time is $M + C + B$ (a sum of memory use, computation and blocking). Because the deadline must be met, the $T \leq M + C + B$ and because it also has to be met in worst case, the $T \leq M + C + B_{max}$. If the inequality is substituted into equation 6.1 and simplified, one would get that $M_{waste} \geq (C + M)\frac{k}{n}$.

The inequality 6.3 quantifies the minimum bandwidth over-allocation with unknown and hence pessimistic blocking (eq.6.2), but optimal task period/deadline. That is the case when no extra slack is left for task completion and in worst case the task finishes just before the deadline. For the tasks with extra slack the waste increases as seen in eq.6.1. The equation 6.1 also shows that the over-allocation increases for tasks whose deadline is smaller than the period. This can be very bad for tasks with tight deadline and a large period. This is because the allocation is made for the whole period, while it has to be high to provide guaranties in short deadline time.

The inequality 6.3 can be interpreted in following way. To provide M "units" of memory, $M\frac{k}{n}$ are wasted because of slack needed to cover blocking uncertainties, while $C\frac{k}{n}$ are wasted because no computation is overlapped with blocking. For small memory transfers, at least some overlapping is likely to be present in a task, and the change in the slot allocation would change the amount of overlapping if it is bad. But for transfers requiring multiple slots, the overlapping is zero (i.e. max blocking) for non-first fragment. So each transfer of multiple slots unavoidably contributes to over-allocation.

We now calculate an optimistic lower bound on memory over-allocation. The best case is when there is no blocking and the computation is totally overlapping with the waiting for the slot. That is the task is a sequence of $\frac{n}{k} - 1$ slots of computation followed by single slot of memory transfer. This leads to the observation that in the best case only $C_{overlapped} = M_{count}(\frac{n}{k} - 1)$ part of computation can be performed without the over-allocation, the rest $(C - M_{count}(\frac{n}{k} - 1))$ will lead to unused memory slots. The M_{count} denotes the number of separate

memory transfers, which is the same as number of first slots.

$$M_{waste}^{opt.anal.} \geq (C - M_{count}(\frac{n}{k} - 1) + M - M_{count})\frac{k}{n}$$

$$M_{waste}^{opt.anal.} \geq (C - M_{count}\frac{n}{k} + M)\frac{k}{n} \tag{6.4}$$

The plain TDM has very simple and efficient hardware implementation, but has limited application for tasks with different memory demands, because the resource is shared equally. Fortunately the flexibility can be easily added by more elaborate slot allocation, i.e. mapping of the time to owner of the slot. The plain TDM is a multiplexer controlled by the modulus-N counter (where N is number of requestors). Arbitrary allocation can be obtained, by inserting some logic or memory between the counter and the multiplexer to map the time to slot owner Maybe drawing would make it obvious to everybody}. The translation can also be performed through a table to create arbitrary slot allocations. Finding the optimal allocation for larger task sets is computationally intractable for the same reason as in the static execution order scheduling. But suboptimal allocation does not violate safety, so heuristics can be used to find one in reasonable time.

6.2.3 RR: Round Robin

The RR is work conserving so it allows better utilization of the memory bandwidth. The utilization is improved because the idle slots are consumed by waiting requesters, and also because there is no fragmentation created by the slots. That is the TDM can only service requests at slot boundary (for fine grained TDM), while RR can start serving requests at any time.

The RR also provides fair sharing of resource most of the time. Requesters get equal share under full load. For random request arriving patterns, the requesters would get a statistically equal share in the long run. If some requesters are active while the other are idle, the RR allows the active ones to consume all the unused time more or less fair. The fairness of the RR depends on the request arriving pattern, and exact fairness is possible only if all requests are pending, because idle requesters are skipped. Such a behavior improves the average case performance because the scheduler adapts to current demand. More demanding requesters have better chances to get bandwidth from those not needing it. But the average case performance is of little use for hard-RT system as reasoned in Section 4.4.1.

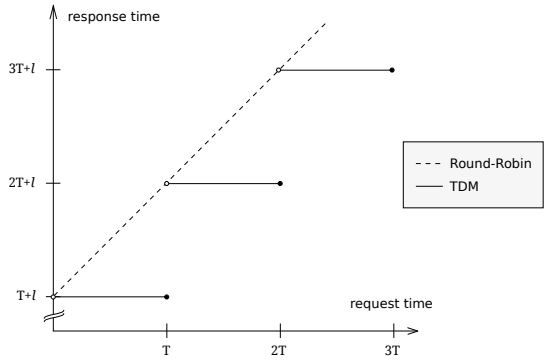

Figure 6.2: The worst case response comparison of TDM and RR arbitration schemes. The time 0 corresponds to requestor's slot in TDM. The slot period T is equal to *slot length* * N (number of requestors). Response is time of waiting to be served and the memory latency l. The RR response graph is a straight line here just as a simplification to show that it is "linearly increasing", in reality it would be the step function with small step of scheduling granularity (for example of one cycle).

The RR is used in RTS because it has bounded response time. Under full load (worst case) its behavior is considered equivalent to TDM. Though under full load, the WCMAT is the same for both TDM and RR, the reference point is different most of the time. For the RR the WCMAT is guaranteed from the time the arbiter sees the request, while for the TDM the biggest WCMAT is only possible at point when request just missed its slot. In other words, timing guaranties provided by RR are always as bad or even worse than TDM (Figure 6.2). The pessimism of RR could theoretically be reduced if the number of total competing requesters at current WCET analysis point can be bounded. Though, if such information would be available, the TDM could also exploit it through configuration of the slot allocation.

Naturally the maximum blocking for RR will be equal to the one derived for TDM in eq.6.2, though for RR this can not be reduced by the analysis only if the number of requesters is reduced:

$$B_{max} = M\frac{N}{1} \tag{6.5}$$

More surprisingly, the better bus utilization of RR does not allow it to reduce the memory bandwidth over-allocation (eq.6.1):

$$M_{waste} = T\frac{1}{N} - M \tag{6.6}$$

This is because the controller must cover the worst case when all the requesters are busy at the same time. The formulas are exactly the same, with only cosmetic difference in bandwidth share. The RR provides guarantees of equal memory share, hence $\frac{1}{N}$ factor is used. While the TDM allows some flexibility in sharing the memory in some other proportions $\frac{k}{n}$.

Finally, the hardware implementation of RR is more complex than of TDM.[1] Incorporating the best effort traffic in bandwidth "unused" by hart-RT tasks is not as simple. It is usually done by allowing some number of non-hard RT tasks to get access to the memory after full round of hart-RT tasks.

An RR based scheme called Dynamic Priority Queue (DPQ) [SRK12b] was proposed to allow not equal sharing of the bandwidth among the requesters. Each requester gets assigned budged in replenishment period and requesters are served RR until they have a budget left. To benefit from the replenishment period, the WCET analysis must know to which period the current request belongs, and how many requests were made in this round already.

Even though we have shown that RR does not provide advantages over the TDM for hard-RTS. Its average case performance is usually slightly better, so it could be used with advantage for non-RT tasks. [PS12] made some benchmark comparison of TDM vs. RR on predictable chip multiprocessor. They used a simplified version of RR with one cycle empty slot per idle requester. The system with simplified RR achieved about 10% higher speed-up than TDM based for some of the benchmarks on 8 cores.

6.2.4 Hybrid TDM-RR

[SRK11] describes a hybrid TDM-RR arbiter which they call Priority Division (PD). The implementation details are not presented, but straightforward implementation would use Programmable Priority Encoder, like the RR. While the RR changes the priority after some requester has been served, the PD changes the priorities relative to the time (like the TDM) independently of who was served last. This way each requester gets a time where it has highest priority, just like in TDM, but if the highest priority requester is idle the next interested requester is granted access.

Actually, the scheme has one peculiarity which actually makes the WCMAT a little bit worse than TDM. It is also an example, when work conserving scheduler

[1] Though the simplifications are possible. For example computation can be performed in multiple cycles, alternatively the single cycle empty slot per requester [PS12].

can provide lower worst-case guaranties, than the same non-work conserving. The problem is caused by the fact that transaction takes several clock cycles. If a request arrives late in a slot, the conserving scheduler would grant the access. This would block the next request for some time, even if it arrives on time in its own slot. The increase in WCMAT is however at most the length of the slot minus one. Also, because the priority update logic is independent from served requests, the delay can not increase more during the next requests. The effect can also be avoided by allowing the requests to only occur at the slot boundary.

The scheme provides all the guaranties, configuration and analysis possibilities of TDM. That is the worst case blocking and bandwidth estimates are the same. Also the same configuration is possible through non equal slot allocation. On the other hand the hardware implementation is more costly. The average case behavior is a little bit different from RR (if the highest priority requests are idle). In principle the sharing among the active requesters is less fair. This is because in RR the same requester can get the access two times in a row only if it is the only requester, while in PD, it can get it as long as there are no higher priority requesters.

The scheme provides good hard-RT guaranties and better average case performance than TDM, thus allowing to combine the hart-RT and other tasks in one scheduler. Thought having the separate TDM based arbitration for hart-RT task and a RR based arbiter scheduled in idle slots would allow more control and fairness for non-hard-RT tasks.

6.2.5 Static Priority

The plain static priority (SP) results in very pessimistic response time for lower priority tasks. This is because WCET analysis need to assume that each single request is blocked by the sum of all the transfers from higher priority tasks, to make this bounded, some rate limiting mechanism is introduced, to prevent the higher priority requester from local burst requests exceeding its allocated bandwidth.

The examples of such arbiters are Credit Controlled Static Priority (CCSP) [Ake10] and Priority Based Budget Scheduler (PBS) [SRK12a]. The PBS performs bandwidth accounting in framed fashion, where the budget for all requesters is reset periodically. In PBS the WCMAT depends on both the time offset in the replenishment period for each memory access and how much bandwidth was used since the beginning of the period. This makes it hard to benefit from the PBS in the WCET analysis, because if such information is not known a pessimistic estimate need to be used. The CCSP fills the budget gradually

by a small fraction so WCMAT is constant if requester does not request more bandwidth than it is allocated.

Under full load (or for longer transfers), all the requesters get their allocated share of the bandwidth, just like it is in TDM. The difference is in the order in which the requesters get their share in the specified time interval. The schemes described here would first serve all the higher priority requesters and low priority requesters would get served at the end of the period. In TDM they would be interleaved. The difference might be an advantage or disadvantage depending on what is needed in the application. The TDM fixes the relation between latency and bandwidth, so it is not possible to efficiently provide low latency for requesters with modest bandwidth requirements. While in SP the lower priority requesters will suffer high latency B_{max}, because they are moved to the end of the period in worst case.

The worst case blocking grows very fast and becomes very bad for systems with many requesters. Also the SP would perform badly for fair bandwidth sharing among requesters with equal demands, because in such case the priorities unnecessarily create low latency for some of the requesters (the one which happened to be lower priority). To reduce the pessimism the distribution of the memory accesses in time for the current task needs to be known. This is for example available when estimating the time to complete a long transfer. This is also available when computation is decoupled from memory access, by introducing some buffers like in data-flow computation. For the conditions of the second example the TDM would behave as good, because buffer makes the computation time dependent on guaranteed bandwidth not guaranteed latency.

Even more pessimism could be removed at the level of schedulability analysis, because this allows using global knowledge about the behavior of the other tasks. By considering that the sum of all blocking experienced by the task could not be greater than the possible memory used by all the higher priority tasks. This however requires the precise model of the tasks memory access in time. Because it is intractable to analyze all the paths, and there is still some variability even in single path (Section 4.2). Probably the WCET of the task could be split into portions where each portion would have may/must properties similar to the cache analysis in WCET. The question has to be answered is if the gain is larger than the pessimism introduced by the approximation.

6.2.6 Dynamic Priority

Intuitively, the best memory access scheduling would be performed by the dynamic priority scheme, where the most critical request is serviced first. The

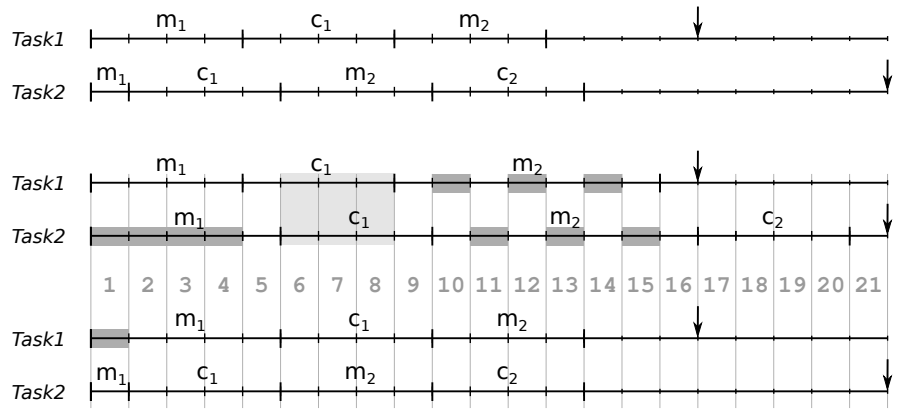

Figure 6.3: Top: single instance of two tasks. Each task will follow single execution path during one instance. Middle: the memory access scheduled according the least laxity first dynamic priority. Bottom: optimal schedule for the same instances.

question is which metric to use to decide whose request is most critical. It seams that the good candidate for estimating what is most critical could be the time slack left for each requester (also known as laxity), but the least laxity first (LLF) scheduling is not optimal as shown in Figure 6.3.

The figure shows some instance (invocation) of two tasks. Both the inputs and the hardware state are fixed for single invocation, and will result in single execution path followed by the task. We now describe the LLF scheduled access (middle part of Figure 6.3). Initially both task want access the memory. The first task has slack of 4 units and the second has 8 units, so the first task will be granted access and the second will be blocked. In slot 5 the first task starts computation and the second gets the memory. During the next 3 slots both tasks will perform the computation, and the memory will be idle. At slot 9 the first task uses the memory while the other finishes computation. The tasks start to interfere again at slot 10. Because they both have the same remaining slack, they will be granted memory access in turns. The example illustrates two inefficiencies of LLF scheduling (both tasks finish earlier in the optimal schedule on the bottom of the figure). The first inefficiency can be seen in slots 6 to 8, where the memory is idle for 3 slots. The idle slots could possibly be used by memory access of some task, which will have to done in future instead. This inefficiency is fundamental to any greedy scheduling algorithm based on single local decision, because the local optimum decision does not necessarily lead to global optimal solution. The other inefficiency manifests itself in slots 10 to 15. Starting from slot 10, both tasks want to access the memory again. The task1

needs 3 more memory access slots, while the task2 4 more. If the task1 would be granted all the required slots without interruption, than the task2 would be blocked by 3 slots and the task1 would be blocked by 4 slots if all is granted to task2. While when the accesses are interleaved like in the example on the Figure, both tasks experience blocking of 3 slots.

However the main problem in using dynamic scheduling here is its analyzability. The analysis at WCET level is not possible, because the worst case blocking depend on the state of all other tasks. Proving that tasks set can meet the deadlines at schedulability level is not trivial either. First of all it involves the same task memory access modeling tradeoffs as mentioned in the previous section. Moreover because of dynamic nature of the scheduler it will be hard to find worst case task interference, while the utilization based schedulability tests needs to be extended. For processor utilization tests, the worst case blocking time has to be taken into account. For memory utilization tests, the worst case memory idle time has to be bounded.

6.3 SDRAM Interface Tradeoffs

In this section we look at the limitation caused by SDRAM interface and options to overcome them.

6.3.1 Access Granularity Tradeoff

The full utilization of the memory bandwidth is only possible by doing large consecutive transfers in the same direction. Thus small accesses lead to wasted memory bandwidth. The Table 5.4 lists how maximum guaranteed bandwidth for random access depends on the size of the transfer for one particular SDR SDRAM memory. For the single-word random access it is as low as 14%, while for 8-word pipelined transfers it is 84.2%. The later generations tend to require larger transfers for the same utilization of the previous generation, because the data rate is doubled with each generation while the timing parameters in nanoseconds do not improve that much. Also the 8 bank devices of DDR2 and DDR3 generations prevent efficient bank interleaving with small transfers by allowing only four activates in t_{FAW} time window. The minimum efficient transfers are also larger when the memory modules are used. For example the standard memory modules for PC are 64-bit wide, so 8-word transfer is actually 64 bytes long.

On the other hand most of the data of fetched big chunks must actually be needed most of the time. For example increasing the burst length of the pipelined transaction from 4 to 8 words increases possible data bus utilization from 50% to 84.2% (Table 5.4), but if half of fetched 8 words are not needed most of the time, the utilization would actually be close to $\frac{84.2}{2}$%. We will now propose the optimization for interleaved transactions which could provide chances for increasing the usage of all data of big operations.

The addresses for interleaved transactions of one memory operation must not correspond to consecutive single block. They can actually be random if they all use different banks. The fact could be exploited by the scratchpad allocation algorithm, because the addresses of managed external memory locations are known. The optimization could also be useful for data caches, because the write-through caches are preferred for RTS, as read miss in write-back cache has a longer worst case latency because the line might need to be written to the memory. So if WCET analysis can know that two addresses are from independent banks, they both can be packed into single slot. Implementation wise the packing would be performed by a write buffer.

The other option is bank privatization similar to one employed in [RLP+11]. Each bank would be assigned to a set of processors. And accesses to banks would be interleaved in the TDM schedule. The processors would than use message passing through NoC to share the data. Alternatively, the TDM allocation can be performed on per bank basis with interleaving allowing to decide which processor need to access each bank in each cycle.

6.3.2 SDRAM Data Rate

The FPGA fabric can not run at frequencies of latest SDRAM generations. The access is performed by using wider data words which are serialized onto the SDRAM data bus. Interconnecting such a wide signals might be problematic for many ports as mentioned in conclusions to Section 7.2.2. Also a caches would need to support wide single cycle transfers, or buffer would be needed per each requester port to perform the adaptation. Alternatively the requesters can be partitioned and few buffers used close to controller as suggested in Section 6.4.

6.3.3 Handling Refresh

The DRAM refresh is required for correct storage of the values, but the operation interferes with regular read/write operations. Even though the refresh is

required to be invoked relatively infrequently, it can contribute significantly to over-allocation if not handled properly. This section discusses possible options and limitation of dealing with the SDRAM refresh in timing analyzable way. The technical background needed for this discussion is provided in 3.6 section.

There are three options:

1. Refresh operations can be grouped together and analyzed at Schedulability Analysis level [BM11]. This option is only effective for SDR generation of SDRAM, because later generations allow to group only 8 refresh operations (Section 3.6.3)
2. They can be invoked individually at a known time, for example in a dedicated TDM slot. One should have in mind the possible $t_{RFC} > t_{RC}$ which would eliminate this option for some memories because it would increase the slot size unnecessary, as mentioned in Section 3.6.2.
3. Or can even run as higher priority periodic operation, which is invoked by the controller and increases WCET of all the tasks appropriately.

Table 6.1: The slots available for memory transfers between the two consecutive reserved refresh slots. This assumes that refresh uses the same slot size as regular operations. The blank fields are for slot size not supported by this frequency.

Slot Size:	7	8	9	10	11	12
50 MHz clk:	54	47	42	38	34	31
100 MHz clk:	110	96	85	77	70	64
125 MHz clk:			107	96	87	80
143 MHz clk:				110	100	92

For option 3. (and 2. with regular schedule) the refresh interference can be pessimistically incorporated into each request. But for architectures where the n cycles delay of certain instruction must contribute n to total execution time, the refresh interference can be applied to whole task's execution time by following recursion [AP01]:

$$WCET_i^{ref} = WCET_{i-1}^{ref} + \lceil \frac{WCET_{i-1}^{ref}}{t_{REFI}} \rceil \cdot t_{RFC}$$

where $WCET_0^{ref} = WCET$. The formula has to be applied recursively, until it stabilizes, because the WCET increased by refresh interference can cause additional interference with the new refresh operations. The formula is approximation because it expects that each refresh will coincide with the memory operation and not some computation. But, the pessimism is smaller than the

one obtained by considering that refresh interferes with each memory operation (if the memory is accessed more frequently than t_{REFI} which is usually the case. For example the SDRAM memory used in the project needs refresh to be issued 8192 times in 64 ms window. This translates to number of cycles available for operation between each refresh presented in the Table 6.1. The cycles are truncated to integer number of full slots for different slot size and clock periods.

It seems that 3. option and the refresh interference incorporated by adjusting WCET calculated without refresh is most simple and efficient option if the architecture is nice enough to allow this option. The small drawback, that the estimated WCET will be safe, but the path hawing this execution might get changed as demonstrated in [AP01] does not seem to cause any problems.

Finally, for interleaved transactions it might be beneficial to perform refresh manually, thus allowing refreshes to be overlapped with the transfers.

6.4 Implications of Hardware Implementation

The hardware cost/speed implications might favor arbitration scheme which is not so efficient "in theory" as other. The T-CREST has a goal to have many processor cores, so scalability might be an issue. Appendix B contains synthesis results of some hardware primitives. From the results it seams that arbitration is speed limited, while the interconnect is area limited, fortunately both limitations can be overcome by constraining the architecture a little bit.

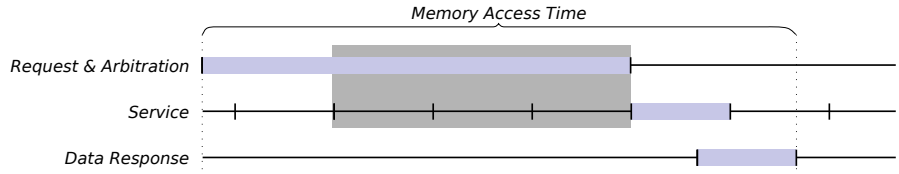

Figure 6.4: Components of memory access time.

The total memory access time can be partitioned into few components as visualized on Figure 6.4. First, some time needed to propagate the memory request to the controller. This takes some fixed minimum latency and additional variable blocking time (marked by gray rectangle on the figure), which is caused by the interference from other tasks or employed scheduling scheme. Next the request is services by the controller. It takes some time until the response data is available in the controller, and additional time to fully propagate the last bit of the

data to the requester. The major contribution to the WCMAT is worst case blocking time, because there are many requesters in the system. The figure hints the memory access pipelining, then the larger latencies of request&arbitration or the response phases do not affect the length of the service slot. This means that in principle the slow arbiters can be used without degrading the system's throughput. This also allows to create more complex, combined arbitration schemes, for example the hard-RT tasks can be scheduled by the TDM, and non-RT and soft-RT by RR but only allowed to use the slot if it is idle.

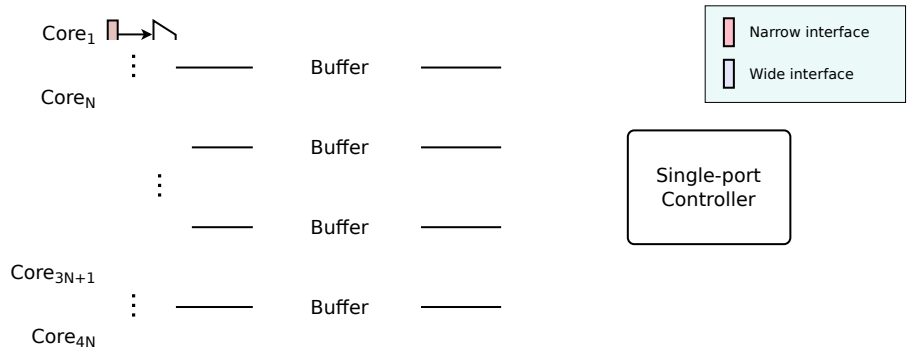

Figure 6.5: Serialization of the interconnect.

The size of the data interconnect can be optimized for the price of reduced flexibility. Because the memory controller can only serve single requester at the time, most of the links are idle. So if the buffers are used near the controller, the data from/to requestors can be serialized during the multiple cycles. The Figure 6.5 shows the conceptual diagram of such architecture. Few multiple narrow interconnects operate in parallel, filling the buffer in multiple cycles. The wide data from the buffer is than used in single cycle by the controller. Such partitioning of the requesters removes some flexibility, for example the two requesters from the same group can not be scheduled one after another without increasing the buffer sizes. Thought for example regular TDM is constrained by design already and will fit such a interconnect naturally, and even narrower link could be used for transfers to the buffers. The optimisation is optional for SDRAM interfaces with narrow words, while might be necessary for wide modules and high data rate SDRAM.

6.5 Discussion

It has been shown that for hard-RTS, the round robin (RR) does not have advantages over time division multiplexing (TDM). This is because better average case performance does not improve the WCET. The memory bandwidth over-allocation is also not reduces by better average memory utilization. Furthermore for TDM, the WCET analysis can derive tighter bounds by exploiting the accesses with known time separation.

The static priority (SP) schemes are not scalable, because the least priority requester will suffer latency proportional to total bandwidth allocation of other requesters.

We propose to use TDM based memory access arbitration. The TDM has number of advantages:

- The interconnect can be optimized.
- For regular arbitration just a modulus counter is required. The configurable slot allocation is also possible. The mapping of time to slots can be performed through a allocation table. Because the table is always read sequentially it can be stored in external memory, and only a small cache used on-chip. The cache can be loaded periodically in a dedicated slot.

Chapter 7

Controller Evaluation

In this chapter we look at the synthesis results of the controllers. In the first part we compare implementation of our controller described in Chapter 5 with some general purpose SDR SDRAM controllers. Next we look at the memory controller for RTS from Technical University of Eindhoven.

7.1 Comparison with Other SDR SDRAM Controllers

We will compare the FPGA synthesis results of our controller with 3 other designs. We first describe each design in short and provide the results at the end of the section.

7.1.1 Altera SDR SDRAM Reference Design

The design is described in [Alt02]. The controller is coded structurally without explicit state machine. Instead the state is distributed into number of internal

control signals and counters which control the multiplexers driving SDRAM signals. The controller is pipelined introducing 4 cycles of additional data latency. The pipeline does not seam to allow multiple outstanding commands to different banks, i.e. the new command is accepted after the previous one is completed (except for Precharge used to interrupt the ongoing full-page Read/Write burst).

The controller provides low-level interface to the SDRAM. Requestor must initialize the SDRAM and must keep track of data latency cycles (i.e. controller does not acknowledge the valid data). The Read/Write requests to the controller are translated into pairs of Activate and Read/Write (with auto-Precharge) SDRAM commands. The requests corresponding to Precharge, Refresh and ModeRegisterSet commands are also available and are used to control the SDRAM initialization sequence. Two more requests are used for runtime configuration of controller's parameters.

7.1.2 Xilinx SDRAM Reference Design

The design is described in [Xil00]. The design is almost 14 years old, and one can see that the synthesis tools had different capabilities and the different hardware description style was used. The controller is coded structurally at a very low-level. The finite state machine (FSM) uses manually specified one-hot encoding for the state register. The Xilinx SRL16 primitives (lookup table used as shift register) are manually instantiated. Each counter is described in a separate entity and the design consists of 9 entities in total. The design provides the requester the same functionality as the Altera design, albeit through slightly different interface. The controller is run-time configurable and the requester must initialize both the controller and the SDRAM. The requests are handled as single burst transaction with auto-Precharge. The requester must count the cycles to know when the valid data should be sampled after read. But the write data is accepted right away and is delayed internally.

The controller uses double frequency clock for SDRAM interface, and communicates the data to/from requester on both edges of the slower clock. The controller seams to be designed to be used from the external chip, because a single 32-bit inout signal is used for address, datain, dataout and part of the command encoding. The controller introduces 8 SDRAM cycles (4 system cycles) of additional data latency during read.

7.1.3 JOP SDRAM Controller

JOP controller [Gra12] was made to enable JOP processor [Sch09a] the access to SDRAM chip on the Altera DE2 board and has hardcoded its timing parameters. Differently to two previously presented designs, the controller performs SDRAM initialization automatically. The controller provides 32-bit SimpCon [Sch09b] interface, but internally uses 16-bit wide SDRAM chip, so some extra buffers and logic is used for this purpose. The two FSM are used, the one handles the SDRAM command sequence, while the other interacts with the SimpCon and assembles/splits the two half-words. The SDRAM address multiplexing is performed in a separate process.

7.1.4 SDR Controllers Synthesis Results

The same synthesis tools setup was used as described in Appendix B, but we limited ourselves to looking at Altera synthesis results for Cyclone II target as it is the FPGA on which the controller is used. The exception was Xilinx reference design, which uses Xilinx specific primitives (LUT shift registers). The Spartan 3 FPGA was used for comparison to have architecture similar to the one for which design was optimized (i.e. the Spartan 2). Our design was re-synthesized for the same target to aid comparison. The default tool settings were used, though for Xilinx synthesis the register packing into IOB was disabled to leave the flip-flops in slices.

Table 7.1: Synthesis results for evaluated SDR controllers. The columns show: clock frequency in MHz, the overall number of Logic-Cells/Slices, the number of Look-Up-Tables and the number of Flip-Flops

Design	F_{max} (MHz)	LC	LUT	FF
Altera	392.77	309	107	284
JOP	207.30	592	457	355
$JOP_{Optimized}$	249.38	308	174	238
Our	221.39	194	126	129
$Our_{SimpCon}$	222.42	272	119	211
$Our_{Optimized}$	349.41	200	127	131
Xilinx Spartan 3	F_{max}	Slices	LUT	FF
$Xilinx_{S3}$	116.20	229	165	293
Our_{S3}	117.79	114	147	130

The synthesis results for three designs mentioned in previous sections and our design from Chapter 5 a presented in Table 7.1. Vertically the table is split in two parts. The first part shows the numbers for Altera Cyclone II target, the second part list numbers for Xilinx Spartan 3. The table has two entries for the JOP design. The numbers for the original design (JOP) were suspiciously large, and we were able to fix the problem by constraining the ranges of the counters ($\text{JOP}_{\text{Optimized}}$). The original design used full range integer types, resulting in the inferring of 32-bit counters. For the purpose of the JOP comparison we have also included the numbers ($\text{Our}_{\text{SimpCon}}$) of our design adapted for the use in JOP system (described in Section 5.6.2) Finally the line $\text{Our}_{\text{Optimized}}$ represents the design after splitting the increment and comparison logic of the refresh counter into separate cycles (by registering the done flag). The initialization counter was also separated and made free running. The optimization is not necessary for our application, because the unoptimized device can run above the required speed. Nevertheless, the high frequency of Altera design, stimulated us to see how big the speed gain of optimisation will be.

The comparison shows that the performance of our simple behavioral controller description is reasonable, and also that some speed gains are possible by simple optimization of critical path (see $\text{Our}_{\text{Optimized}}$ vs. Our). We would like to make some comments to the obtained results:

- Altera design is clearly optimized for speed, so the high frequency is not surprising. The design is pipelined and the control logic is distributed always depending on just few bits. The critical path is on some wide multiplexer used to initialize a delay counter according the configuration register.
- The higher FF count of Altera design comes from pipelining, especially of the 32-bit wide data signal. All the input/outputs are also additionally latched, whereas Our design expects the requesters to hold the address and data stable. The Xilinx design uses many FF to register the input and additional to store it in few places, because the address/configuration/data all use single bus.
- Even though the maximum clock frequency in Xilinx and Our designs are almost the same, the Xilinx controller supports higher SDRAM frequencies, because the SDRAM interface is operated with double frequency. The same trick could probably be applied to speed up Our design if needed, thought the frequency might decrease because of possible introduction of critical path at the shorter period paths.

7.2 A Look at TU/e DDR3 Controller

In this section we make a closer look at predictable controller for DDR3 SDRAM from Technical University of Eindhoven. The TU/e controller is used here because its source code was provided for the early integration of controller with the Patmos processor which was performed by the author of this thesis. The report of the integration work is included in the Appendix D. We first present the TU/e controller in more details. Next we discuss the synthesis results.

7.2.1 The TU/e controller

The controller is based on work published in [Ake10] and was introduced in Section 2.1. The controller allows using any arbiter with bounded response time, and the Round Robind is used in this particular implementation.

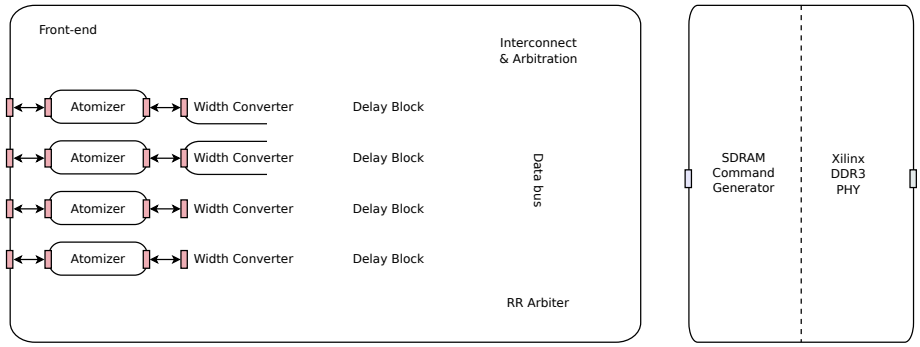

Figure 7.1: High level view of the controller.

The organization of the controller is shown on Figure 7.1. There are two main parts, the front-end allowing multiple requestors to use a single-port controller and the back-end which is the controller itself.

Front-end performs certain request transformation and buffering on each requester port, before the arbitration and multiplexing is performed. The atomizer breaks the long requests into requests of elementary size. The width converter adapts between 32-bit data word of the user interface and 128-bit data interface of the low-level DDR3 controller. This 4× deserialization/serialization is needed, because the DDR3 SDRAM interface runs at twice the frequency of the controller and transfers two words per cycle. The delay block delays the read data until the worst case latency. The delay makes the response time repeatable

Page contains confidential material and has been removed.

Page contains confidential material and has been removed.

Page contains confidential material and has been removed.

Chapter 8

Conclusions

This chapter summarizes the thesis and suggests possible improvements to the subject.

8.1 Contributions and Findings

There are two main contributions of this work:

- The open source SDR SDRAM controller has been created. Its initial integration into two RTS platforms (T-CREST and JOP) was performed and tested.
- The different options of memory access scheduling for the T-CREST platform have been investigated. The analysis included estimates of their RTS efficiency and the hardware implementation feasibility. The analysis conclusions are presented in Section 6.5

Some of the most interesting observations made:

- For hard-RTS, the round robin (RR) does not have advantages over time division

multiplexing (TDM), whereas WCET bounds can be made tighter with TDM.

- The static priority (SP) arbiters like CCSP and PBS are not scalable for WCET analysis because the least priority requester will suffer from latency proportional to the total bandwidth allocation of other requesters.
- The memory access timing analysis performed at WCET level suffers from fundamental limitations in reducing memory bandwidth over-allocation. The local worst case required bandwidth has to be allocated for the whole task's execution period.

8.2 Suggestions for Future Work

During the project, many questions arose which could not be answered because of a limited time frame. Some suggestions for future work are:

- Look at the programming models. The way the external memory is used should be re-thought in the context of available inter-core communication and on chip memories. This might lead to some programming model specific optimizations in the access arbitration to external memory.
- Further explore the possibilities of performing memory access analysis at the schedulability level. The precision and feasibility of the approach will depend on task's memory demand modeling.
- The hard-RTS are by definition safety critical and might benefit from error correcting code (ECC) memories. The ECC codes are stored for whole memory word width, so writing a smaller portion requires reading the word, updating a part, recalculating the ECC and storing a word to memory. Some of the decisions in controller design could be affected by this constraint.

Appendix A

Source Code Access

The source code created during the Thesis project is accessible online through T-CREST git repositories:[1]

Controller's repository (git clone git://github.com/t-crest/sdram.git):
vhdl/sdr_dram.vhd - The controller's source
vhdl/sdr_dram_opt.vhd - The optimized version mentioned in Section 7.1.4
vhdl/sc_sdram_wrapper.vhd - The SimpCon wrapper (Section 5.7.2.2)
simulation/sdr_sdram_dma_controller_tb.vhd - The test bench
simulation/vsim_sdr_sdram - Makefile, simulation scripts and settings

The integration with the Patmos (git clone git://github.com/t-crest/patmos.git):
Patmos integration with our controller (Terasic/Altera DE2-70)
quartus/altde2-70_sdram - The project directory
vhdl/top/patmos_de2-70sdram.vhd - Top entity
vhdl/io/patmos_io_sdram.vhd - I/O port mapping
vhdl/core/patmos_sdram.vhd - Patmos Core
ise/ml605_edk/pcores/dma_controller_dtl_v1_00_a - The I/O device connecting the processor and memory controller

[1] We use a convention in this chapter, that the hyperlink would invoke the on-line version, while the command is printed for fetching the file locally

c/test_sdram.c - Integration test program
c/test_sdram_full.c - Integration test program
Patmos integration with TU/e DDR3 controller (Xilinx ML605)
ise/ml605_edk - The main dir, with the project and make file
ise/ml605_edk/pcores/dma_controller_dtl_v1_00_a - The I/O device connecting the processor and memory controller
ise/ml605_edk/pcores/patmos_sdram_v1_00_a - The Xilinx Platform Studio component for the Patmos processor
asm/test_sdram.s - Integration test program
asm/test_sdram2.s - Integration test program
asm/test_sdram3.s - Integration test program
ise/ml605_edk/pcores/dma_controller_dtl_v1_00_a - The I/O device connecting the processor and memory controller

Appendix B

Scalability of Primitives for Arbitration and Interconnect

This Appendix provides results of hardware cost and speed estimates for some primitives what would be needed to implement the interconnect and arbitration for multi-port controller.

The primitives examined were: static and programmable priority arbiters; multiplexers; binary encoder and decoder. The size parameterized hardware descriptions were used to obtain synthesis figures for several size points. Most examples were adapted from [Alt11] and [Chu06] with minor modifications. The ripple implementation of programmable priority arbiter was coded from scratch. The circuits are purely combinatorial, but to derive the propagation delay for the Alter tools the wrappers were created to register the inputs and outputs. The circuit sizes are reported for the version without those extra registers.

All the circuits were synthesized for both Xilinx Virtex 6 and Altera Cyclone II FPGAs by using the standard FPGA vendor toolchains (Xilinx ISE version 14.2 and Altera Quartus version 12.0) with the default options. These particular FPGAs were chosen because they were used during the project, this also allowed to test both tool vendors, as well as two different FPGA architectures. The Virtex 6 is a high-end chip using architecture optimized for high performance. The logic fabric features the 6-input lookup tables (LUT), dedicated multiplexers, dedicated xor gate and carry chains for fast adders [Xil12]. The Cyclone II has

less aggressive architecture optimized for lower cost. The logic is mapped into 4-input LUTs and carry chains for ripple adders are available [Alt07].

The default settings where used to instruct the tool in deriving results with balanced size and speed optimizations. The tool does not do aggressive speed optimizations which could increase the area enormously. The synthesis with area optimization option produced similar results, except for ripple implementation of programmable priority arbiter with broken combinatorial path (pp_ripple_r) on Xilinx. The numbers for this design are included in the table separately.

The names used in the tables:

- decoder: binary decoder [Chu06];
- encoder: binary encoder [Chu06];
- mux: 1-bit wide multiplexer (described behaviorally);
- sp_1hot_adder: 1-hot encoded static priority arbiter, implemented by a bit scan through carry chain [Alt11];
- sp_chu: binary encoded static priority arbiter [Chu06];
- pp_1hot_adder: 1-hot encoded programmable priority arbiter, implemented in long carry chain [Alt11];
- pp_double_sp: binary encoded programmable priority arbiter implemented by combining the two masked static priority arbiters [Chu06];
- pp_ripple: 1-hot encoded programmable priority arbiter implemented by propagating the priority from programmed input through all unused lower priority inputs. Contains a well behaving combinatorial loop.
- pp_ripple_r: pp_ripple with the combinatorial loop broken by a flip flop. Because of this needs two cycles to output the result.

Table B.1: Look-up tables usage for different size of primitives (CycloneII).

Primitive	8	16	32	64	128
decoder	8	20	40	98	145
encoder	3	10	18	43	90
mux	5	10	21	42	85
sp_1hot_adder	15	31	63	127	255
sp_chu	5	14	33	105	272
pp_1hot_adder	24	48	96	192	384
pp_double_sp	5	72	159	359	838
pp_ripple	19	39	77	156	310
pp_ripple_r	17	38	74	156	308

Table B.2: The length of critical path (in ns) for different size of primitives (CycloneII).

Primitive	8	16	32	64	128
decoder	1.105	1.776	2.027	2.336	2.226
encoder	1.065	1.746	2.085	2.479	3.357
mux	2.000	2.537	3.432	3.957	4.721
sp_1hot_adder	2.375	3.063	4.363	7.040	12.104
sp_chu	1.876	2.348	3.110	4.951	7.046
pp_1hot_adder	3.194	4.750	7.258	12.677	23.310
pp_double_sp	1.647	4.949	5.762	7.203	11.257
pp_ripple	6.061	10.586	21.053	51.308	7.275
pp_ripple_r	3.219	4.486	8.762	17.584	36.337

Table B.3: Look-up tables usage for different size of primitives (Virtex6).

Primitive	8	16	32	64	128
decoder	4	8	32	34	132
encoder	3	6	15	36	77
mux	2	4	10	21	42
sp_1hot_adder	5	24	48	96	192
sp_chu	4	9	44	103	201
pp_1hot_adder	24	48	96	192	384
pp_double_sp	4	12	155	328	686
pp_ripple	8	16	32	64	128
pp_ripple_r	14	29	60	116	227
pp_ripple_r(area.opt)	7	15	31	63	127

Table B.4: The length of critical path (in ns) for different size of primitives (Virtex6).

Primitive	8	16	32	64	128
decoder	1.157	1.346	1.404	2.015	2.044
encoder	1.111	1.553	2.117	2.105	1.957
mux	1.505	1.694	2.251	2.761	3.064
sp_1hot_adder	1.759	2.183	2.495	3.119	4.367
sp_chu	1.619	2.563	6.410	10.989	13.264
pp_1hot_adder	2.339	2.577	3.201	4.523	6.945
pp_double_sp	1.619	2.543	12.067	15.498	20.734
pp_ripple	3.963	6.615	11.919	22.527	43.743
pp_ripple_r	3.682	5.405	7.341	8.876	10.242
pp_ripple_r(area.opt)	2.972	5.394	9.386	17.370	33.338

Page contains confidential material and has been removed.

Page contains confidential material and has been removed.

Page contains confidential material and has been removed.

Page contains confidential material and has been removed.

Appendix D

Patmos and TU/e SDRAM Controller Integration Report

This appendix includes the report of the integration work performed by the author during the thesis project. It describes the early integration of patmos processor with draft version of memory controller delivered by TU/e university.

The integration sub-project involved many small thing. First the Xilinx ML605 FPGA board supported by the provided SDRAM controller had to be setup on the Linux server, which demanded a little bit of intervention into the Xilinx toolchain install process, which did not work out of the box. Next the Patmos processor had to be tested on the Xilinx FPGA (the processor was developed on an Altera FPGA board). Minor modifications to the code were made, to allow inferring of correct primitives by the toolchain of both vendors. An EDK component was made for Patmos, because SDRAM controller test system was provided as EDK project. The I/O based interface to the SDRAM was developed to allow seamless integration with the current state of Patmos pipeline. The integration was tested in hardware with assembler based test programs. Finally the report describing the integration work was written, and follows in this Appendix.

D.1 Overview

The current version of controller has following properties:

- controller interfaces the DDR3 memory on Xilinx *ML605* FPGA development board;
- only access of multiple of 64 byte blocks are supported;
- 4 processor ports with proprietary DTL interface are supported;
- the controller can not run reliably at full speed of 200 MHz because of the slight timing violation.

The current version of the Patmos does not contain caches yet so it was decided to test the integration by using simple processor controlled I/O device. The device provides single cycle memory mapped interface. The processor issues the memory operations and pools the device if completion status is needed. The device translates processor's memory requests into DTL transactions and provides them to the controller. The setup is shown on Figure D.1.

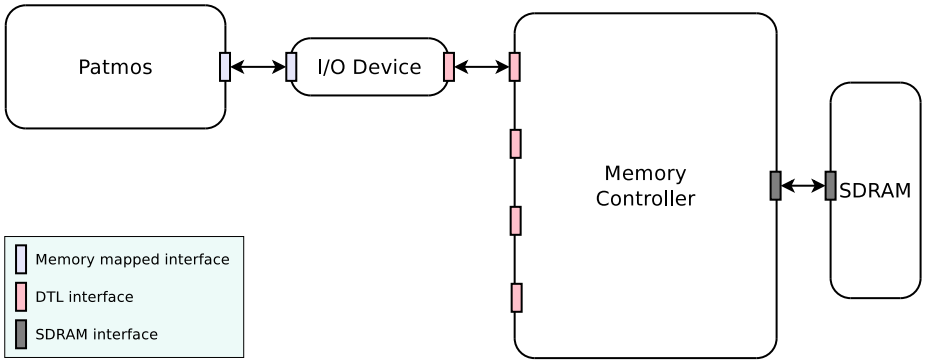

Figure D.1: Overview of the integration

D.2 Controller DTL Interface

DTL interface consists of three logical signal groups:

- **command group.** Used to issue the read/write command. This also specifies the operation address and size.
- **read group.** Receive the requested data.
- **write group.** Transmit the requested data.

Each group has a *valid* / *accept* signal pair for the handshake. The transfer happens in the cycle when both signals are high. The read and write group use the *last* signal to signal the end of transmission. VHDL signals for reference:

D.3 I/O Device Interface

The single cycle interface[1] is provided to the Patmos. So the processor can interact with the device without a stall. The following I/O registers of the device are visible to processor:

- **data buffer**: The data input/output happen through a buffer in the device. The buffer is 64 bytes, because it is a size of single memory transfer supported by current controller.
- **address**: Defines the address for the controller's memory operation.
- **command**: A read/write of 64-byte memory block is initiated.
- **status**: Processor pols the status to find out if block read/write operation has been completed.

D.3.1 Address Mapping

The register of I/O device are mapped to Patmos address space starting at `0xf0000300` base address. The register addresses are word aligned, i.e. two least significant bits are not used. Here's the table of offsets from the base address:

Offset (binary)	Description
`0000000-0111100`	16 words of **data buffer**
`1000000`	**address** to load/store the block in memory
`1000100`	**command** (during write): LOAD_BLOCK = 0; STORE_BLOCK = 1 **status** (during read): READY = 0; BUSY = 1

[1] currently the interface is combinatorial, i.e. the responce is provided in the same cycle. But this can of course be pipelined if needed.

D.3.2 I/O Device Implementation

D.3.2.1 State Machine

Simple Moore style state machine is used to control the interaction. Following states are present:

- **ready**: The device is not engaged in any transactions with memory controller. Data buffer and address register is accessible for processor. The processor can also initiate a block load/store command.
- **read_cmd**: The device issues the DTL read request.
- **read_data**: The device receives a word. The counter is used in this state to address the data buffer.
- **write_cmd**: The device issues the DTL write request. This can be merged with **write_data** state to start the transfer one cycle earlier.
- **write_data**: The device transmits a word. The counter is used as in the **read_data** state.

D.3.2.2 Data Buffer

The single port asynchronous memory is used. The port is connected to patmos interface in **ready** state and to DTL controller interface otherwise (the counter of transferred words managed by state machine is used as an address).

D.3.2.3 Device Status

The state determines the status of the device when queried by processor. If the device is not in **ready** state, the issued block load/store operation has not been completed yet and processor has to wait.

D.4 Testing

D.4.1 Simulation

First the VHDL code of the I/O device was tested by performing ModelSim simulation. The testbench emulating both the processor requests and memory with

DTL interface was created. The same testbench was used in both behavioral and post-translate code of the device.

D.4.2 Pre-Integration Experiments

The provided controller test project was using two leds to output its status. The one led was showing SDRAM initialization completion and the other one was asserted when the traffic generator has successfully completed the test. Before the integration was started, the provided test project was extended with the uart transmitter and small logic to display the status of the leds on the serial terminal. This was done to test the integration of custom component into the XPS project and to use the board remotely.

D.4.3 Assembly Tests of The Whole Integration

Once the integration was performed and bit file successfully generated, the assembly programs were written to test the SDRAM access from patmos:

- `asm/test_SDRAM.s`: Tests the data buffer of the I/O device without accessing the SDRAM. 16 ASCII characters are written to the buffer. Next they are read back and output onto the serial terminal.
- `asm/test_SDRAM2.s`: Test the SDRAM by using only the first word of the data buffer. First word of each SDRAM block is written with different value. The values are read back and checked afterwards. The discovered errors are reported immediately by outputting 'E' character. At the end of the test 'OK' is printed (or '##' if errors were detected).
- `asm/test_SDRAM3.s`: This is test analogous to test2, but all the words of the SDRAM are written and checked.

All the tests wait for character input before get started. This is used to wait for completion of the memory initialization. The address range for test 2 and 3 is configurable through register r10. The tests are also described in source file comments.

D.4.4 On Chip Signal Analysis With ChipScope

It was initially not possible to get positive results from running assembly tests. ChipScope was used to inspect the I/O device signals at both patmos and con-

troller interfaces. Inspection showed that I/O device was receiving the requests from the patmos, and correct transactions were performed with the controller. As a result the assembly programs were reviewed to find and fix errors.

D.5 Notes About The Tools

D.5.1 ssh: Remote Use of the Board

Most of the work was performed remotely through text terminal over ssh. `screen` program was used to allow multiple terminals on single connection. Port forwarding was used to access the licenses for Xilinx software and ChipScope server. Following `~/.ssh/config` file was used to specify a shortcut host `imm` with all the needed settings:

```
HOST imm
    Hostname sshlogin.imm.dtu.dk
    User s081553
    ForwardX11 yes
    Compression yes
    LocalForward 2100 eda1:2100
    LocalForward 2101 eda1:2101
    LocalForward 8080 socwiki:80
    LocalForward 50000 procell:50000
    LocalForward 50001 procell:50001
```

D.5.1.1 remote configuration through cse_server

The `cse_server` can be started on the remote computer with FPGA board connected. The iMPACT/ChipScope GUI can be run locally with remote cable selected.[2] The ssh port forwarding was used as mentioned earlier, because the remote computer is behind the firewall.

D.5.1.2 batch configuration of the FPGA

The FPGA configuring by batch mode iMPACT (console) was used when the .bit file was already on the server. This was for example handy when only

[2] http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/pim_p_remote_configuration.htm

assembly code was changed. The following wrapper script was created for convenience[3]:

```
#!/bin/bash

#Get the full path to bit file argument
case "$1" in
  /*.bit) BITFILE="$1";;
   *.bit) BITFILE="$PWD/$1";;
   *)
        echo "  Usage: $0 BITFILE"
        echo "Programs the file specified as argument into the ML605 using
 the impact in batch mode"
        exit 1
   ;;
esac

# setup the impact
#source /opt/Xilinx/14.2/LabTools/settings64.sh /opt/Xilinx/14.2/LabTools/
export PATH=/opt/Xilinx/14.2/LabTools/LabTools/bin/lin64:$PATH
LD_LIBRARY_PATH=/opt/Xilinx/14.2/LabTools/LabTools/lib/lin64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH

# setup the temporary dir
TMPDIR=‘mktemp -d‘
trap ’rm -rf "$TMPDIR"’ EXIT
IMPACT_SCRIPT=$TMPDIR/program.cmd
cd $TMPDIR

#create the impact script with given .bit file path
cat > "$IMPACT_SCRIPT" <<EndOfTemplateFile
setMode -bs
setCable -p auto
setcablespeed -speed 12000000
identify
assignFile -p 2 -file "$BITFILE"
program -p 2
quit
EndOfTemplateFile

# start the impact
impact -batch "$IMPACT_SCRIPT"
```

D.5.2 xps: Xilinx Platform Studio

After failures to use controller from ISE, it was decided to convert patmos to XPS component. Because more integration work would have to be done in

[3]This is hardcoded for this board, because it assumes that FPGA is at position 2 in JTAG chain

future, it was experimented with doing so without a XPS GUI.[4] This turned out to be possible and quite well functioning GUI-less flow was discovered.

Following EDK documentation was relevant:

- EDK Concepts, Tools, and Techniques: introductory document, but not very useful as it focuses on GUI method, which happened to work very badly with given example project.
- Embedded System Tools Reference Manual: contains description of command line (no GUI) invocation of XPS and synthesis tool (`platgen`).
- Platform Specification Format Reference Manual: contains description of file format, which is valuable reference when manual modifications of the file are performed.

In general, the content of the files is intuitive and copy-paste, learn by example approach of creating component can be used. There is however one tricky moment with version number, which will result in "cannot find MPD for the core in any of the repositories" error. The cause of the error is hard to spot because files seam to be there. There are two distinct version kinds, and both has to match for component to be discovered (see next sections):

- MHS (specification) version: `2.1.0` in provided project
- component version: this is arbitrary user defined number with format `N.NN.L`, for example `1.00.a`.

D.5.2.1 Component description

Custom components are looked up from `pcores` subdirectory of the project. The component's directory has following directory structure:

```
pcores/dma_controller_dtl_v1_00_a
├── data
│   ├── dma_controller_dtl_v2_1_0.mpd
│   └── dma_controller_dtl_v2_1_0.pao
└── hdl
    └── vhdl
        ├── dma_controller_dtl.a.vhd
        └── dma_controller_dtl.e.vhd
```

[4]The peripheral creation wizard method was used before to connect the I/O device to XPS project. This method did not look to be promissing as it required significant amount of clicking and respecifying signals. Also it was impossible to make connection between some of the ports because the GUI would filter the signals according the classess, so it was not possible to even make connections present in the original project.

```
└── dma_controller_dtl.p.vhd
```

Notice the two distinct version kinds (component/mhs) mentioned before:

- `<COMPONENT>v<COMP_VERSION>/data/<COMPONENT>v<MHS_VERSION>.mpd` file describes the interface of the component.
- `<COMPONENT>v<COMP_VERSION>/data/<COMPONENT>v<MHS_VERSION>.pao` file describes library name and sources constituting the component.
- `<COMPONENT>v<COMP_VERSION>/hdl/vhdl/` directory contains the sources. The `pao` file contains specifies source paths relative to this directory.

The provided project had a convention of having separate entity, architecture and package files (the `.e.vhd`, `.a.vhd` and `.p.vhd` suffixes in previous listing). However this turned out not to be required by the XPS. This was also a little bit confusing, because the (default) values for generic parameters where specified in multiple places: `mhs` (top level), `mpd`, `.e.vhd` and `.p.vhd` files.
The content of `mpd` file is intuitive. Here's example line from `pao` file:

```
lib dma_controller_dtl_v1_00_a dma_controller_dtl.a.vhd
```

- The library name (second word on the line; `dma_controller_dtl_v1_00_a` in this example) in `pao` file should match the components directory name.
- The one of the specified sources should contain the top level entity for the component. It should have the same name as the component (without version number; `dma_controller_dtl` in this example).

D.5.2.2 Top level (*.mhs file)

The mhs file is a top level, it describes external ports (should be matched with the ucf file), component instantiations and their interconnection (port mapping). The format of the file is intuitive. Here's some abridged example (`#` starts comment, `...` is used to denote omission):

```
...
# This version should be used in the MPD and PAO file names:
PARAMETER VERSION = 2.1.0
...
# Example of external ports: PORT <UCF_NAME> = <INTERNAL_WIRE_NAME>, ...
PORT fpga_0_rst_1_sys_rst_pin = sys_rst_s, DIR = I, SIGIS = RST, RST_POLARITY = 1
...
PORT txd = txd, DIR = O
PORT rxd = rxd, DIR = I
...
# Example of component instantiation:
BEGIN dma_controller_dtl
 # Unique instance name:
 PARAMETER INSTANCE = dma_controller_dtl_0
 # This version should be used in components directory name:
 PARAMETER HW_VER = 1.00.a  # dma_controller_dtl_v1_00_a in this example
 ...
 # Values for generic parameters if default values from MPD file need to be
 # overridden
 PARAMETER DMA_ADDR_WIDTH = 4
```

```
 PARAMETER DMA_DATA_WIDTH = 32
 ...
 # This is special multi-signal wire (equivalent of record in VHDL), has to
 # be defined in MPD file
 BUS_INTERFACE DTL_OUT = dma_controller_dtl_0_DTL_OUT
 # Regular signal port mapping: PORT <SIGNAL_NAME> = <INTERNAL_WIRE_NAME>
 PORT mtl_clk = raptor_0_Clk_200 MHz_bufg_o
 PORT mtl_rst_n = nRst_Res
 PORT dma_addr_special_i = dma_addr_special_i
 ...
END
...
```

The internal wires are not declared, arbitrary name can be used in `PORT` lines, the ports which use the same name of the wire get connected. The `BUS_INTERFACE` signals does not need to be specially coded in VHDL, they are just defined in MPD file as a set of regular signals.

Signal transformations

- Concatenation of wires is the only transformation allowed in `PORT` mapping, for example:

  ```
  PORT three_bit_port = one_bit_wire & two_bit_wire
  ```

- The opposite operation (bit slice) of the wires is not allowed and special `util_bus_split` XPS component need to be used for this.
- The special `util_vector_logic` XPS component need to be used to perform boolean functions on the signals (for example invert).

D.5.2.3 XPS project file, makefile creation and download of proprietary controller code

The `system.xmp` is a plain text project file. When it is opened with `xps` GUI, the top-level .mhs file and used cores are verified. Finally the `system.make` and `system_incl.make` files are created together with some additional directories for settings and compilation results. The `-nw` option of `xps` for starting without the GUI is used to perform these operation automatically.
The wrapper `Makefile` is created to automate the whole process:

```
#default target, this shows help from system.make
all:

%: system.make force
	make -f system.make $@

force: ;

system.make: pcores_proprietary
	@echo Creating Makefile from XPS project...
	# this open the project and creates the makefile
	echo exit | xps -nw system.xmp
```

```
pcores_proprietary:
        @echo Downloading SDRAM controller XPS cores...
        scp -r tipca.imm.dtu.dk:/home/edgarlakis/TUE_memctrl/pcores $@

program: bits
        program_ML605 implementation/system.bit
```

The rules in this Makefile will automatically launch the project to create the XPS project build makefiles and use them to build requested target. The `pcores_propritary:` target additionally downloads the proprietary code for memory controller which can not be published on the git.

D.5.3 data2mem: Initialize the Patmos Instruction Memory in bit File

The synthesis of whole system take some time. Because boot loader is not available yet, the **data2mem** [5] tool was used to replace the patmos code in final bit file. This works with block ram memories only, so initially the code in `vhdl/generated/patmos_rom.vhd` was modified to use recommended code for block ram inferring. Later, as size of patmos instruction memory grew, the block ram was inferred from unmodified code.[6]
In addition to .bit file the data2mem needs a file describing the memory content and placement of the block ram on the FPGA. They are described in next subsections. Then correct input is ready, the patmos instruction memory can be initialized to content of `instruction.mem` file with following command:

```
data2mem -bm patmos.bmm -bd instruction.mem -bt original.bit -o b updated.bit
```

D.5.3.1 Block RAM placement (.bmm file)

The following .bmm content was used:

```
ADDRESS_SPACE fet_rom_evn RAMB36 [0x00000000:0x000011FF]
    BUS_BLOCK
        patmos_top_0/patmos_top_0/fet_rom_evn/Mram_data_mem1 [35:0] PLACED = X0Y21;
    END_BUS_BLOCK;
END_ADDRESS_SPACE;
```

The word after `PLACED =` identifies the block ram on a device to which the instruction memory is placed, because PAR uses heuristic algorithms this changes for each synthesis and must be updated.[7]

[5] http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/data2mem.pdf

[6] Patmos uses dual (even/odd) memory for instruction loading, but in current version the odd part was optimized away so only single block ram is used for instruction memory.

[7] `bitgen` can update the file automatically if .bmm file is given to `ngdbuild`, but XPS does not know about our interest of this memory and creates empty .bmm file each time.

The placement of the instance can be discovered by loading post routed design with PlanAhead/FPGAEditor, alternatively .ncd file can be converted to text representation with the `xdl` tool from which the needed information can be found without clicking the GUI. Following command sequence was used after bit file generation:

```
cd ise/ml605_edk
xdl -ncd2xdl implementation/system
grep 'inst.*fet_rom' system.xdl
```

D.5.3.2 Memory content (.mem file)

The initialization content is described in plain text file, like this:

```
@0
0 00000000
0 000A000F
```

The line with `@0` defines the offset, next comes the data. All the numbers are hexadecimals. The block ram used is actually 36 bits wide and the first hexadecimal character in each line is used to initialize 4 bits unused by patmos. All the spaces separating the data are actually ignored by data2mem and were used for readability.
The patmos assembler was used to get the binary code, which was later converted to .mem file by small C program:

```
./bin/paasm asm/test.s - | bin2mem > instructions.mem
```

Here's the C source of bin2mem.c for reference:

```
#include <stdio.h>
#include <stdint.h>
#include <byteswap.h>

int main(int argc, char *argv[]) {
        int i;
        int cnt = 0;
        int32_t val;

        printf("@0\n");
        while ((i=read(0, &val, sizeof(val))) > 0)
        {
                printf("0 %08X ", __bswap_32(val));
                if (++cnt % 8 == 0)
                        printf("\n");
        }

        return 0;
}
```

The bytes are swapped because the program was run on little endian (intel) processor and .mem file expects most significant bits first.

D.5.4 Assembly Labels

The tools for programming patmos are in early stage and are not particularly user friendly. Writing larger assembly program is error prone. The labels for branch instructions are not supported, so branch offsets need to be updated each time instructions are added/removed before the branch target. Small perl script was written to help mitigate the problem.
First version of the script would add/udate the comment containing the instruction number, so that offsets for the target could be seen in source code. Next the code was improved to also add the automatically resolved labels. The example fragment of assembly source produced by the script:

```
        addi    r0 = r0, 0;  # first instruction not executed          #0
        addi    r12 = r0, 0; # r12==error count                       #1
        addi    r5 = r0, 15;                                          #2
        sli     r5 = r5, 28; # r5==uart base                          #3
        addi    r6 = r5, 768;# r6==SDRAM base                         #4

# wait_start:   # Output '?' and wait for any key press
        addi    r1 = r0, 63; # '?'                                    #5
        swl     [r5 + 1] = r1;                                        #6

#poll_stdin:
        lwl     r1 = [r5 + 0];                                        #7
        addi    r2 = r0, 2;                                           #8
        and     r1 = r2, r1;                                          #9
        cmpneq  p1 = r1, r2;                                          #10
        (p1)    bc      7;   #l:poll_stdin                            #11
                addi    r0  = r0 , 0;                                 #12
                addi    r0  = r0 , 0;                                 #13
```

The comments with the numbers at the end of the lines count the instructions. The commented at the start of the lines are the labels (`#wait_start:` and `#poll_stdin:`). The comment `#l:poll_stdin` after branch instruction at line `#11` denotes that target of the branch should be the instruction after `#poll_stdin:`. This is correctly resolved as `7` by the script.
The source code of the script for reference:[8]

```
#!/usr/bin/perl -T
#Insert the line number comments in patmos code
# also resolve the labels

my $ic = 0; #instruction count
my %labels = ();

while(<>) {
        $line = $_;
        if ($line =~ /^#?\s*(\w+):/) {
                #label or label comment
```

[8] only backward labels are resolved because script uses single pass through standard input, but this was enough for such temporary tool

```
                $labels{$1} = $ic
        }
        if ($line =~ /^\s*(\(\w+\))?\s*(bc)\s+(\d+)\s*;\s*#\s*l:(\w+|\+\d+)/) {
                # branch with label
                my ($pred, $instr, $target, $label)  = ($1, $2, $3, $4);
                my $new_label;
                if ($label =~ /\+(\d+)/) { # relative branch
                        $new_label = $ic+1+$1;
                } else {
                        $new_label = $labels{$label};
                }
                #print "#$pred,$instr,$target,$label,$new_label\n";
                if (!defined $new_label) {
                        $new_label = "$target.Unkown"
                }
                $line =~ s/(\s)$target(\s*;)/\1$new_label\2/;
        }
        if ($line !~ /^\s*(\w+[:])?\s*(#|$)/) {
                # line containing the instruction (non-empty, non-comment line)
                $line =~ s/(\s*# ?[0-9]+\s*)?\n//;
                printf "%-60s\t#%d\n", $line, $ic++;
        } else {
                # leave other lines unchanged
                print $line;
        }
}
```

D.5.5 ChipScope

ChipScope was used to diagnose the problems of I/O device by probing the signals while the design was running on the FPGA.
Before ChipScope analyzer GUI can be used, two ChipScope cores ICON (Integrated Controller) and ILA (Integrated Logic Analyzer) must be included in the design.[9] There are several ways to put them into the design, the PlanAhead tool was used to connect ChipScope cores to Post-PlaceAndRoute design.[10]
The board was used remotely on relatively slow Internet connection, ChipScope server feature was very handy. It allowed to run ChipScope GUI on local machine and communicate with the `cse_server` running remotely. The ChipScope built-in wave browser seemed to be not very usable, so the data was exported in .vcd format and inspected from GTKWave.

[9] http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/chipscope_pro_sw_cores_ug029.pdf
[10] http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/PlanAhead_Tutorial_Debugging_w_ChipScope.pdf

D.6 Encountered Problems and Conclusions

D.6.1 XPS Project Integration with ISE

D.6.1.1 Import Generated VHDL Files in ISE Project

- The "Netgen" was failing to find child components. This was confusing error as the correct hierarchy was shown in ISE, but synthesis of child files was never tried to be performed because of "$\text{black}_{\text{box}}$" attributes.
- The compilation would proceed after attributes were removed, but it would fail because standard XPS components are not available in ISE. They where also not available from core-generator. It is probable that they could be specified as some switch to one of the tools. Though XPS uses different synthesis tool (platgen) so this might also be impossible.

D.6.1.2 Use XPS Project as Component (black-box) in ISE Project

- This approach was documented and should work. However the method was failing because XPS had put the IOB at the port signals. Older version of XPS would have a setting where one could select this behavior, I assumed that newer version of the tool was doing this automatically. The solution was to modify the EDK project file manually. The bit file was generated as a result, but it was not working[11].

D.6.2 Clock Frequency and Failing Timing Constraints

Failing timing constraints were observed during test with provided traffic generator test project. The issue was discussed with Martin and it was agreed on reducing the clock frequency, but it was forgotten to do so in final integration. Quite some time was waisted on trying to run examples on overclocked patmos, because some code would actually run correctly.
The issue was remembered and clock frequency reduction test was first performed on provided controller example with traffic generator. Next it was replicated on patmos integration. All the generated clocks were slowed down twice to preserve clock relation in current code and avoid new clock-domain crossing. The fix made patmos run much better.

D.6.2.1 Errors in the Assembly and Conditional-Store Bug

Because current version of the patmos did not support call/return, the assembly test code happened to grow very fast. It became hard to know that was wrong and to spot mistakes. Reduction of the code to some minimal interaction with the SDRAM happened not to help, because of the encountered patmos bug. The

[11] it might be just clock frequency, so if this method of integration is preferred the issue can be revised

predicated (conditional) store instruction would always be executed. Unluckily this instruction was used to report errors over serial terminal, so it seamed that the simplest tests of SDRAM were failing.

D.6.3 Conclusions

The integration work was successful with following useful results:

- the integration was performed and tested;
- learned new tools (XPS, data2mem, PlanAhead, ChipScope, GTKWave);
- patmos bug discovered.

Unfortunately 1/3 of time (if not more) was spend debugging. Ideally part of this time waste could be avoided. Looking retrospectively, doing things differently from technical point of view could help slightly:

- Could probably ask Sahar or Martin about the patmos toolchain to exploit it better. For example C compiler or maybe simulator to debug the SDRAM test programs.
- The larger fifo buffer in the serial code could be used for the test. This way the serial status polling code could be removed from test programs making them shorter and containing only SDRAM test relevant code.

Maybe these and even more improvement ideas could come during the project (and not after it) if it would be possible to deal with following non-technical issues:

- The task was estimated overoptimistically, for example lengthy assembly programs should not be expected to work on processor/tools out-of-the box in such early stage of development.
- It is very easy to spend lots of time looking for the problem because of limited observability. Might be that the overtime was indirectly negatively contributing to overall time.

D.7 Appendix: Source Code Location

The project of the integration and source of assembly programs is in patmos git repository and can be retrieved with following command:
`git clone git://github.com/t-crest/patmos.git`
After fetching the source, invoke `make` without arguments in `ise/ml605_edk` directory to get more instructions. The default make rule will guide you for rest of the steps. This would require access rights to proprietary code of TU/e memory controller.
Some notes on using the board were put into the SoCwiki of IMM institute at DTU:
http://socwiki/doku.php?id=xilinx_ml605

Bibliography

[AGR07] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In Soonhoi Ha, Kiyoung Choi, Nikil D. Dutt, and Jürgen Teich, editors, *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2007, Salzburg, Austria, September 30 - October 3, 2007*, pages 251–256. ACM, 2007. URL: http://doi.acm.org/10.1145/1289816.1289877.

[Ake10] Benny Akesson. *Predictable and Composable System-on-Chip Memory Controllers*. PhD thesis, Eindhoven University of Technology, February 2010. URL: http://www.es.ele.tue.nl/~kakesson/publications/pdf/akesson-dissertation.pdf.

[Alt02] Altera Corporation. *SDR SDRAM Controller White Paper*, ver. 1.1 edition, August 2002. URL: http://www.altera.com/patches/ref_design/ref-sdr-sdram-vhdl.zip.

[Alt07] Altera. *Cyclone II Architecture, Cyclone II Device Handbook*, 3.1, feb 2007 edition, 2007. URL: http://www.altera.com/literature/hb/cyc2/cyc2_cii51002.pdf.

[Alt09] Altera. *SDRAM Controller Core, Quartus II Handbook Version 9.1 Volume 5: Embedded Peripherals*, v9.1 edition, November 2009. URL: http://www.altera.com.cn/literature/hb/nios2/n2cpu_nii51005.pdf.

[Alt11] Altera. *Advanced Synthesis Cookbook*, 2011. URL: http://www.altera.com/literature/manual/stx_cookbook.pdf.

[AP01] Pavel Atanassov and Peter Puschner. Impact of dram refresh on the execution time of real-time tasks. In

Proc. IEEE International Workshop on Application of Reliable Computing and Communication, pages 29–34, Dec. 2001. URL: http://www.vmars.tuwien.ac.at/php/pserver/extern/docdetail.php?DID=808&viewmode=paper.

[BLL+11] Dai N. Bui, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Jan Reineke. Temporal isolation on multiprocessing architectures. In Leon Stok, Nikil D. Dutt, and Soha Hassoun, editors, *DAC*, pages 274–279. ACM, 2011. URL: http://doi.acm.org/10.1145/2024724.2024787.

[BM11] Balasubramanya Bhat and Frank Mueller. Making DRAM refresh predictable. *Real-Time Systems*, 47(5):430–453, 2011. URL: http://dx.doi.org/10.1007/s11241-011-9129-6.

[But11] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer, 2011. URL: http://books.google.dk/books?id=h6q-e4Q_rzgC.

[BW01] A. Burns and A.J. Wellings. *Real-Time Systems and Programing Languages: Ada 95, Real Time Java and Real Time Posix*. International Computer Science Series. Addison-Wesly, 2001. URL: http://books.google.dk/books?id=0_LjXnAN6GEC.

[CBRJ12] Elliott Cooper-Balis, Paul Rosenfeld, and Bruce Jacob. Buffer-on-board memory systems. In *ISCA*, pages 392–403. IEEE, 2012.

[CDKM02] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, 2002. URL: http://books.google.dk/books?id=oD5mH26tkewC.

[CHO12] Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen. What is a timing anomaly? In Tullio Vardanega, editor, *WCET*, volume 23 of *OASICS*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[Chu06] P.P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley, 2006. URL: http://books.google.dk/books?id=gVd2yeFHshUC.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011. URL: http://doi.acm.org/10.1145/1978802.1978814.

[EKL+09] Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proceedings of IEEE International Conference on Computer Design (ICCD 2009)*. IEEE, 2009. URL: http://www.jopdesign.com/doc/pret_iccd.pdf.

[Ern04] Rolf Ernst. Mpsoc performance modeling and analysis. In *Multiprocessor Systems-on-Chips: Systems on Silicon*. 2004.

[Gra69] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[Gra12] Julian Grahsl. JOP DRAM support for Altera DE2 board (source code), 2012. URL: https://github.com/jop-devel/jop/blob/master/vhdl/memory/sc_dram16.vhd.

[GRW11] Daniel Grund, Jan Reineke, and Reinhard Wilhelm. A template for predictability definitions with supporting evidence. In Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, March 18, 2011, Grenoble, France*, volume 18 of *OASICS*, pages 22–31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011. URL: http://dx.doi.org/10.4230/OASIcs.PPES.2011.22.

[IE06] Nicholas Jun Hao Ip and Stephen A. Edwards. A processor extension for cycle-accurate real-time software. In Edwin Hsing-Mean Sha, Sung-Kook Han, Cheng-Zhong Xu, Moon hae Kim, Laurence Tianruo Yang, and Bin Xiao, editors, *EUC*, volume 4096 of *Lecture Notes in Computer Science*, pages 449–458. Springer, 2006.

[IS407] Integrated Silicon Solutions, Inc. *16Meg×16 256-MBIT Synchronous DRAM*, March 2007. URL: ftp://ftp.altera.com/up/pub/datasheets/DE2-70/Memory/sdram/42S83200B-16160B.pdf.

[JES08] JEDEC Solid State Technology Association. *Double Data Rate DDR SDRAM Standard, (JESD79F)*, February 2008. URL: http://www.jedec.org/sites/default/files/docs/JESD79F.pdf.

[JES09] JEDEC Solid State Technology Association. *DDR2 SDRAM Standard, (JESD79-2F)*, November 2009. URL: http://www.jedec.org/sites/default/files/docs/JESD79-2F.pdf.

[JES10] JEDEC Solid State Technology Association. *Low Power Double Data Rate (LPDDR) SDRAM Standard, (JESD209B)*, February 2010. URL: http://www.jedec.org/sites/default/files/docs/JESD209B.pdf.

[JES11a] JEDEC Solid State Technology Association. *Low Power Double Data Rate 2 (LPDDR2), (JESD209-2E)*, April 2011. URL: http://www.jedec.org/sites/default/files/docs/JESD209-2E.pdf.

[JES11b] JEDEC Solid State Technology Association. *Wide I/O Single Data Rate (Wide I/O SDR), (JESD229)*, December 2011. URL: http://www.jedec.org/sites/default/files/docs/JESD229.pdf.

[JES12a] JEDEC Solid State Technology Association. *DDR3 SDRAM Standard, (JESD79-3F)*, July 2012. URL: http://www.jedec.org/sites/default/files/docs/JESD79-3F.pdf.

[JES12b] JEDEC Solid State Technology Association. *DDR4 SDRAM Standard, (JESD79-4)*, September 2012. URL: http://www.jedec.org/sites/default/files/docs/JESD79-4.pdf.

[JES12c] JEDEC Solid State Technology Association. *Low Power Double Data Rate 3 SDRAM (LPDDR3), (JESD209-3)*, May 2012. URL: http://www.jedec.org/sites/default/files/docs/JESD209-3.pdf.

[JNW08] Bruce L. Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008. URL: http://www.elsevierdirect.com/companion.jsp?ISBN=9780123797513.

[MT12] Inc Micron Technology. Sdram product catalog, 2012. URL: http://www.micron.com/products/dram/sdram.

[Pit09] Christof Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009. URL: http://www.vmars.tuwien.ac.at/php/pserver/extern/download.php?fileid=1659.

[PKP09] Peter Puschner, Raimund Kirner, and Robert G. Pettit. Towards composable timing for real-time programs. In *Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems*, STFSSD '09, pages 1–5, Washington, DC, USA, 2009. IEEE Computer Society. URL: http://dx.doi.org/10.1109/STFSSD.2009.26.

[PNC] M. PAOLIERI, E.Q.U.I. NONES, and F.J. CAZORLA. Timing effects of ddr memory systems in hard real-time multicore architectures: Issues and solutions. URL: http://people.ac.upc.edu/fcazorla/articles/ACMTECS-2013-HRT-memcntrl.pdf.

[PQCV09] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Systems Letters*, 1(4):86–90, 2009. URL: http://doi.ieeecomputersociety.org/10.1109/LES.2010.2041634.

[PS12] Wolfgang Puffitsch and Martin Schoeberl. On the scalability of time-predictable chip-multiprocessing. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 98–104, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2388936.2388953.

[RLP+11] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. PRET DRAM controller: bank privatization for predictability and temporal isolation. In Robert P. Dick and Jan Madsen, editors, *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9-14 October, 2011*, pages 99–108. ACM, 2011. URL: http://doi.acm.org/10.1145/2039370.2039388.

[RWT+06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[Sch09a] Martin Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Number ISBN 978-1438239699. CreateSpace, August 2009. Available at http://www.jopdesign.com/doc/handbook.pdf. URL: http://www.jopdesign.com/doc/handbook.pdf.

[Sch09b] Martin Schoeberl. *SimpCon - a Simple SoC Interconnect*, 2009. URL: http://www.jopdesign.com/doc/simpcon.pdf.

[Sch09c] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009. URL: http://www.jopdesign.com/doc/ca4rts.pdf.

[Sch12] Martin Schoeberl. Is time predictability quantifiable? In *International Conference on Embedded Computer Systems (SAMOS 2012)*. IEEE, 2012. URL: http://www.jopdesign.com/doc/tpquant.pdf.

[SRK11] H. Shah, A. Raabe, and A. Knoll. Priority division: A high-speed shared-memory bus arbitration with bounded latency. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*,

pages 1 –4, march 2011. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5763319.

[SRK12a] H. Shah, A. Raabe, and A. Knoll. Bounding WCET of applications using SDRAM with priority based budget scheduling in MPSoCs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 665 –670, march 2012. URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6176554.

[SRK12b] Hardik Shah, Andreas Raabe, and Alois Knoll. Dynamic priority queue: An SDRAM arbiter with bounded access latencies for tight WCET calculation. *CoRR*, abs/1207.1187, 2012. URL: http://arxiv.org/abs/1207.1187.

[SSP+11] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, 2011. URL: http://www.jopdesign.com/doc/patmos_ppes.pdf.

[TW04] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004. URL: http://dx.doi.org/10.1023/B:TIME.0000045316.66276.6e.

[Wan05] D.T. Wang. *Modern dram memory systems: performance analysis and scheduling algorithm*. PhD thesis, 2005. URL: http://drum.lib.umd.edu/handle/1903/2432.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. URL: http://doi.acm.org/10.1145/1347375.1347389.

[WGR+09] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009. URL: http://dx.doi.org/10.1109/TCAD.2009.2013287.

[Xil00] Xilinx. *Synthesizable High Performance SDRAM Controller, Application Note XAPP134*, v3.1 edition, February 2000. URL: http://www.xilinx.com/bvdocs/appnotes/xapp134.pdf.

[Xil11] Xilinx. *Virtex-6 FPGA Memory Interface Solutions*, ug406 march 1, 2011 edition, 2011. URL: http://www.xilinx.com/support/documentation/ip_documentation/ug406.pdf.

[Xil12] Xilinx. *Virtex-6 FPGA Configurable Logic Block*, ug364 (v1.2) february 3, 2012 edition, 2012. URL: http://www.xilinx.com/support/documentation/user_guides/ug364.pdf.