Performance evaluation of recent DRAM architectures for embedded systems

Performance Evaluation of Recent DRAM Architectures for Embedded Systems

Matthias Gries, Andreas Romer
Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology Zurich
CH-8092 Zurich, Switzerland
email: {gries,aromer}@tik.ee.ethz.ch

TIK-Report No. 82
November 12, 1999

Abstract

The growing gap between processor speed and memory access time becomes more and more a performance limiting factor in modern computing systems. Therefore, DRAM manufactures try to improve memory throughput by developing new memory interfaces around basically unchanged memory cores. This report focuses on embedded systems, i.e., systems which utilize less cache space and fewer memory hierarchy levels than ordinary PC or workstation systems due to costs, area, and power dissipation restrictions. Therefore, embedded systems particularly depend on the performance of the underlying main memory system. Hence, two recent DRAM architectures, widely-used SDRAMs and the next generation memory Direct RDRAM, are investigated in this report. Performance gains are revealed that can be achieved by exploiting features of recent memory interfaces with simple enhancements of current embedded memory controllers. Different approaches for memory access schemes are investigated by simulation of the DRAM architectures and the memory controller together with an out-of-order issue, superscalar CPU model running various applications. The simulations lead to the following results: using RDRAMs instead of SDRAMs improves the performance of the system by up to 21%*. However, in many cases this difference in speed can be compensated by an optimized memory controller design that exploits the pipeline and bank structure of recent DRAMs. An analysis of address traces shows that the described improvements better consider the locality characteristics of the applications because expensive (in terms of latency) row misses in the current memory bank can be shifted to cheaper accesses to other memory banks.*

Contents

1 Introduction

Since memory chips have always been optimized for capacity and not for access speed, they have become the main performance bottle-neck of computing systems [3, 28, 5]. Memory chip manufacturers have tried to bypass the weak performance of the memory core by designing complex memory interfaces which isolate the behavior of the slow memory core from the fast interconnections between the memory chip, a memory controller, and a central processing unit. This has led to a variety of memory interface implementations which essentially are based on the same memory core technology and functionality.

Recently, most manufactures of personal computer technology have decided to switch from synchronous DRAM (SDRAM) to Direct Rambus DRAM (RDRAM). The RDRAM interface is not simply another step of SDRAM interface improvement, but needs a complete redesign of the main memory system. The Rambus approach for improving memory utilization is to use narrow buses, so-called channels, at very high clock rates. In traditional SDRAM based main memory systems, several chips are needed to fit the width of the memory bus. Thus, these chips cannot be controlled concurrently. A single RDRAM chip, however, spans already the whole Rambus channel. Hence, RDRAM chips can be controlled concurrently. Moreover, more internal memory banks can be found in an RDRAM than in an SDRAM.

The question now arises whether computing systems will be able to exploit the new features introduced by RDRAMs. In this report, we concentrate our investigations on embedded systems where cache sizes and the number of main memory DRAM chips are kept small. For this class of systems we compare SDRAMs and RDRAMs for a number of applications, ranging from rather small CPU benchmarks to large real-world programs. We also investigate different memory access schemes which can be implemented by designing appropriate memory controllers.

Related work

Detailed cycle-true simulators of CPUs have been used for a variety of investigations in order to explore future processor systems together with their memory subsystem. These studies either focus on the influence of different CPU architecture features and cache designs on the memory behavior of the whole computing system using standardized SPEC [26] CPU benchmarks [5, 8, 27] and particular applications [24, 27] or concentrate more on the workload generated for the memory subsystem by certain applications [1]. All these studies have in common that due to large second level caches the impact of the main memory is supposed to be low. Thus, the main memory is simulated by a simple model which only consists of two to three access delay parameters. Embedded systems however often do not use second level caches and even the size of the first level cache is limited because of cost, power, and area constraints. Therefore, these systems are in particular dependent on the main memory design. Hence, we model the main memory system more precisely. Especially, the functionality of the memory controller is emphasized since the memory controller is a component which can be redesigned, patched by external circuitry, or at least configured for an embedded application. This procedure however is not feasible by the user for the embedded CPU. Accordingly, the CPU architecture is fixed for our simulations.

Enhancements of memory controllers such as stream buffers [17, 9] and configurable complex address remapping functions [6] are too complex to be introduced in embedded

memory controllers in the near future. The concepts need adjustments at the compiler, may generate additional run-time overhead for reconfigurations, and may introduce additional delay for applications which do not show a regular memory access pattern due to further scheduling and control stages within the memory controller. We thus restrict our study to simple memory controller architectures and enhancements which can be found in current personal computing systems and workstations. These enhancements will be available for embedded processors in the near future.

Recent RAM surveys [21, 18, 15] focus on the functionality, the architecture, and typical applications of dynamic RAMs. However, they do not provide performance measurements under realistic workloads. An analysis of different DRAM architectures and efficient access schemes has been combined with performance investigations by simulation of a complex CPU model together with main memory by Cuppu et al. in [7]. However, their objective differs from ours in that they simulate a workstation class computer with larger caches, longer cachelines, more superscalar units, and a higher clock frequency than embedded systems usually have. Cuppu et al. concentrate on an analysis and comparison of different DRAM types in order to clarify where time is spent during a main memory access. In contrast to that, we are interested in the impact of the memory controller on the performance of the whole computing system. We restrict our studies to the two most recent DRAM types, but simulate more precise DRAM and memory controller models.

This report is organized as follows: in the next section, the functionality of a memory controller is briefly described and the characteristics of SDRAMs and RDRAMs are pointed out. Section 3 then outlines the chosen CPU simulator and the extensions implemented in order to simulate a CPU model together with detailed main memory models. Section 4 finally presents the different simulation settings and the results obtained. The report concludes in section 5.

2 Functionality of a memory subsystem

2.1 Functionality of a memory controller

A minimal computing system is displayed in Fig. 1 consisting of a central processing unit (CPU), a memory controller, and memory chips forming the main memory. The memory controller is responsible for the translation of read and write requests of the CPU into control signals for the DRAM. Linear addresses must be translated into two-dimensional addresses for rows and columns which are used inside the memory chip. In addition, the controller must satisfy the timing requirements of the memory chip because the timing of control signals is not checked by the DRAM itself. In the worst-case, the memory controller has to stall the issue of new requests until the DRAM is again capable of accepting read or write accesses.

The memory bus may be subdivided into three buses, namely for data, control signals, and addresses. In some systems, combinations of these signals are multiplexed. The control signals at a certain point of time can be seen as a memory instruction by which the memory chip is programmed. Embedded CPUs often use a memory controller which has been integrated into the CPU core. Personal computing systems rather use an external controller through a front-side bus for more flexibility in the system design.

Memory controllers without additional operation queues are not able to reschedule read and write operation requests from the CPU. Most of the memory controllers currently found in PC and embedded systems belong to this class. Thus, the order of operation

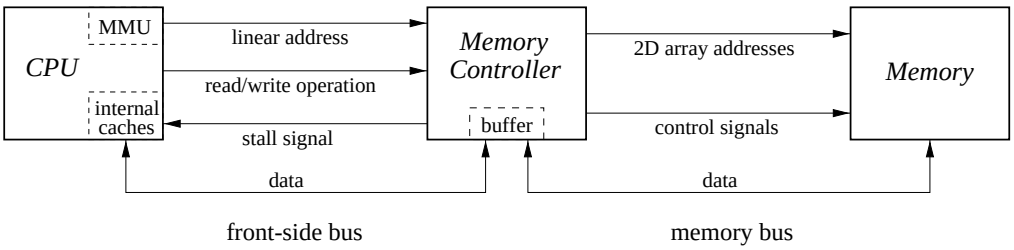

Figure 1: *A minimal computing system.*

requests issued by the CPU is unchanged. However, the latency of a memory operation can be reduced by using heuristics which for instance leave memory rows opened as long as possible in order to save some precharge and activate operations if further memory accesses appear to the already opened memory row. This strategy is called an *open-page* policy. In contrast to that, a *closed-page* policy precharges an active memory row as soon as possible. This method is employed if page or bank changes appear frequently due to data accesses with weak locality. Moreover, the controller may buffer at least an additional memory request of the CPU. Such a controller is able to exploit the pipelined memory interface of modern DRAMs by overlapped processing of the current and the next pending access. Requests cannot be rescheduled, but the sequence of control signals can be shortened.

The format of the memory accesses issued by the CPU depends on the CPU architecture and is usually set at start-up time. The size of an access equals the size of a cache line so that a single burst transfer is sufficient to exchange the contents of a cache line.

Looking at current main memory implementations which use SDRAMs as memory technology it can be seen that the front-side bus and the memory bus usually run at the same clock speed. Moreover, the bus widths of both buses are equal in most cases. Consequently, the memory controller just buffers incoming signals until corresponding control signals can be set. In the RDRAM case however, the front-side and the memory bus differ in the clock frequency and the bus width. Thus, the controller must perform a kind of serial to parallel conversion and vice-versa in order to cope with the different clock rates.

Finally, the memory controller may implement simple schemes to map physical address ranges to other regions by taking the memory organization into account. Hence, shorter access latencies may be achieved by less exchanges of memory pages due to address permutations.

2.1.1 Features of current memory controllers

In embedded CPUs the memory controller is often an integrated module and provides much fewer features than stand-alone controllers. Under the condition to support synchronous as well as asynchronous RAMs, most 32 bit embedded CPUs only support a reduced feature set of SDRAMs. For instance, the *MPC8xx* controller series by Motorola only use closed-page mode. The *Coldfire* controller by Motorola and the *V832* by NEC provide logic for the recognition of page hits in open-page mode. However, only a single open page is supported per memory chip (Coldfire) and for the whole main memory system (V832) respectively. Moreover, successive memory accesses cannot be processed concurrently and only a limited set of timing parameters can be adjusted. In comparison to that, the

80960RN by Intel and the *MPC8240* by Motorola recognize page hits on up to four open pages per chip (80960RN) and for the whole memory system (MPC8240) respectively. Again, overlapped processing is not offered. Beyond that, the *SH-4* by Hitachi is able to overlap accesses from the CPU core and the DMA controller. Accesses from the same device (CPU or DMA controller) cannot be overlapped since the memory controller only looks at the current access in process and the next pending request of the other device.

Stand-alone controllers are not only used in embedded systems but also in personal computing systems. Stand-alone memory controllers need an additional front-side bus which adds more flexibility in system design but also generates additional delay, usually one to two bus clock cycles per access. The *MPC106* by Motorola uses two open-page counters for the whole memory system in order to recognise page hits and to perform overlapped processing for consecutive accesses. The *VRC5074* by NEC and the *440BX* by Intel can control up to eight open pages for short page hit latencies and overlapping memory control sequences. The *21174* by Digital supports up to 24 open pages and uses a four bit history predictor for each open page in order to determine whether the row should be kept open or closed. Recently introduced memory controllers such as the *ApolloPro133* by Via Technologies and the *751* by AMD are able to not only look at the next pending request but also reschedule several read and write requests for overlapping memory control sequences. The AMD 751 buffers up to 16 pending read and six pending write requests and schedules them according to data dependencies and page hit or miss behavior.

Tab. 1 summarizes the features of current memory controllers which use SDRAMs. The support for a closed-page (*c-p*) activation policy, for an open-page policy together with the maximal number of open pages (*o-p/#pages*), and for overlapped processing is shown.

chip name	**c-p**	**o-p / #pages**	**overl.**
embedded CPUs with integrated memory controller			
Hitachi SH-4	yes	yes / 4	yes
Intel 80960RN	no	yes / 8	no
Motorola Coldfire	yes	yes / 1	no
Motorola MPC8xx	yes	no	no
Motorola MPC8240	yes	yes / 4	no
NEC V832	yes	yes / 1	no
stand-alone memory controllers			
Intel 440BX	yes	yes / 32	yes
Motorola MPC106	yes	yes / 2	yes
NEC VRC5074	yes	yes / 8	yes
workstation class stand-alone memory controllers			
AMD 751	yes	yes / 24	yes
Digital 21174	yes	yes / 24	yes
Via Tech. ApolloPro133	yes	yes / 8	yes

Table 1: *Features of current memory controllers.*

2.2 Functionality of recent DRAMs

SDRAMs and RDRAMs have several features in common. They both have synchronous interfaces for the memory bus. These interfaces isolate the main memory cell arrays from the signals of the memory controller. The control interface has a command pipeline for the memory banks in the DRAM. Nevertheless, it is normally not possible to transfer a memory instruction on each clock cycle for interleaved processing of the parallel memory

banks because the banks have to share the memory bus and an input/output buffer pair. Moreover, the RAM chip itself does not check for data collisions on its internal data path.

Read and write accesses are usually performed in burst operation mode, that is, data words on successive addresses are transferred without the need of additional address transfers by the memory controller. An arbitrary access is a two-step process. The corresponding row of a memory array must be addressed before a particular column is accessed. The contents of the chosen row are then held by the *sense-amplifiers*. Thus, the sense-amplifiers can be seen as a single row cache of the corresponding memory bank. Accesses to that particular row are fast and just need the column access time. Accessing another row however requires exchanging the contents of the amplifiers and is slow, since the current row in the sense-amplifiers must always be precharged before another row can be addressed. Modern DRAMs consist of several internal memory banks, each with its own row of sense-amplifiers. In the context of DRAM organization, an internal memory array row is often called a memory *page*.

2.2.1 SDRAM

SDRAM is currently the most often used synchronous DRAM type. Especially, the RAM modules according to the PC 66/ PC 100 SDRAM specifications [12, 13, 11] by Intel and the imminent PC 133 standard are popular.

SDRAM chips are currently most common in 64 Mbit, 128 Mbit, and 256 Mbit capacities distributed over up to four memory banks. Up to 4 KByte of information can be offered concurrently in the sense amplifiers of all banks. The memory bus is subdivided into there separate buses for data, address, and control signals, which all use the rising edge of the clock as reference.

The burst length can be set for read and write accesses up to a full page burst. The refresh overhead can be kept small since the maximal refresh interval is 64 ms and several banks can be activated and precharged concurrently with a single control instruction.

For the implementation of large memory spaces SDRAMs are available on standardized modules on which several memory chips are mounted. The modules provide a fixed data bus width. The most popular modules called DIMM (Dual In-line Memory Module [14]) offer a 64 bit wide memory bus. Different configurations are possible, e.g. a 64 bit wide module for SDRAMs may consist of eight RAM chips with an eight bit organisation or of four chips with an 16 bit organisation, see Fig. 2 for the 16 bit variant. In this example, the shaded memory banks of the distinct chips are controlled by the same signals and therefore cannot be accessed individually. Besides, there are limitations which consider the maximal count of memory modules that can be supplied by a single memory controller due to load and timing constraints since several modules are usually connected in parallel to the memory bus.

2.2.2 RDRAM

RDRAM is a recent memory specification developed by Rambus Inc. RDRAMs will be initially available in 64 Mbit, 128 Mbit, and 256 Mbit sizes. The 64 Mbit variant uses 16 memory banks, the bigger ones 32 banks. The rows of an internal memory bank have a size of 2 KByte looking at 256 Mbit devices. However, a sense amplifier row covers just half of the row entries. Therefore, sense amplifier rows must be shared between adjacent memory banks in order to cache the information of a complete memory array row in two sense amplifier rows. This is why just half of the available memory arrays can be kept

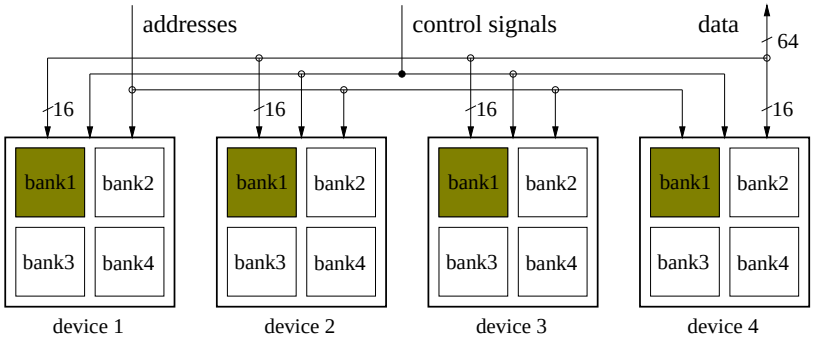

Figure 2: *SDRAMs connected in parallel to the bus on a DIMM.*

activated concurrently in the best case. The sense amplifiers are however still able to cache 32 KByte of information in a 256 Mbit chip.

An RDRAM uses a single bus for addresses and control instructions as well as a separate bus for data transfers. Both edges of the clock are used on both buses. The transmission of all kinds of information is started at the falling edge of the clock and needs four clock cycles (eight clock edges) for completion. Thus, the burst length is fixed to eight data words of 16 bit. Since even control signals need several clock cycles for transmission, RDRAMs are often referred to as DRAMs which use packets for communication. The maximal clock frequency of the buses is 400 MHz.

RDRAMs use write buffers. The memory controller must take care of the data consistency of the write buffer and the sense amplifier contents. That is, the contents of the buffer must be transferred to the corresponding sense amplifiers with the help of a so-called *retire* instruction packet before the appearance of any subsequent read or precharge instructions for the same memory location. However, a retire instruction can be skipped if a read or a write instruction to another RDRAM chip is issued. In this case, the write buffers of all other devices are retired automatically. Moreover, a write instruction after another write to the same device also retires the write buffers automatically. Nevertheless, the additional retire commands may delay subsequent operations. Consider a situation where a read operation follows a write operation to the same memory address. In this case, the read operation must be delayed until the write buffers are in a retired state.

A RDRAM device must be refreshed twice as often as an SDRAM. The minimal row active time of 50 ns (20 clock cycles) as well as a column address strobe delay of 30 ns (12 clock cycles) are typical for a DRAM core. Therefore, the minimal row active time is a limiting factor for fast arbitrary accesses through the "overclocked" RDRAM input-/output interface. Furthermore, there are no instructions for the concurrent activation or precharge of parallel memory banks within an RDRAM. Thus, the overhead for the refresh of the whole chip is greater than for an SDRAM.

RDRAMs are also available on memory modules, so-called RIMMs specified by Rambus Inc. Up to 16 RDRAMs can be combined on a single RIMM which provides a 16 bit wide memory bus. A single RDRAM device already spans the 16 bit memory bus. Thus, all memory banks on a RIMM can be controlled individually since the chips on a RIMM are connected in parallel to the buses (as DIMM are in the SDRAM case). Several RIMMs are interconnected in a serial fashion. Up to 32 RDRAMs distributed over up to three RIMMs form a Rambus channel. The different interconnection schemes are displayed in

Fig. 3.

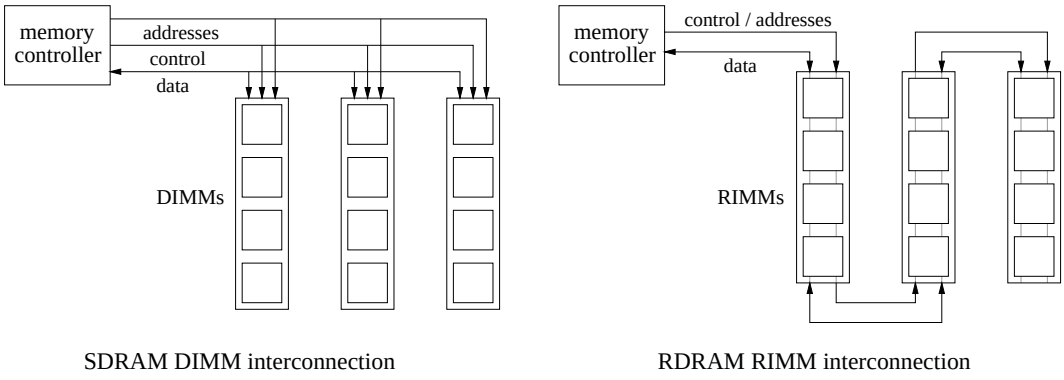

Figure 3: *Interconnection schemes.*

3 Simulation environment

3.1 Simplescalar tool set

3.1.1 Alternatives

SimICS SimICS [16] is an instruction-set simulator of the Sparc V8 instruction set architecture. It thus only simulates the functional behavior of the system and not the cycle-true timed behavior. Instructions are interpreted in program order, each with a duration of one simulated clock cycle. That is, neither pipelines nor any other architecture specific items are considered. The simulator can execute programs in the Sun Solaris binary format directly. Thus, an operating system is not required but may be also simulated in addition. SimICS can be used in the fashion of a debugger; programs can be halted, disassembled, statistics can be generated, and profiling is enabled on different granularity levels. The slowdown by a factor of 30 to 40 due to simulation versus execution on real hardware is at a moderate level. The simulator comes in binary format. Source code is only available for some extensions.

SimOS SimOS [23] provides models of the MIPS R4000, R10000, and Digital Alpha processor families. In addition to the CPU, caches, multiprocessor memory buses, disk drives, ethernet, consoles, and other devices are simulated. There are functional and timed cycle-true processor models available for simulation. The source code (written in C) is freely available. An operating system running on the simulated CPU is required. If applications run under the chosen OS on a real CPU, no further porting steps are required in order to run the simulation. The slowdown due to simulation depends on the granularity of the chosen CPU model and and is up to a factor of several thousand which is in the normal range for a timed cycle-true simulation.

RSIM RSIM [19] is a timed cycle-true simulator for custom, shared-memory multiprocessor systems and uniprocessors. In particular, multiple issue, out-of-order issue and completion, branch and address speculation, and non-blocking loads and stores are modeled. The simulated CPU architecture is similar to the Mips R10000. The instruction

encoding however is similar to the Sparc V9 ISA [25]. Programs have to be ported to run under RSIM for multiprocessor and shared memory investigations. Only user-level code can be simulated, system calls are mapped onto the simulation host system via calls to a custom application library. The source code (written in C/C++) is freely available.

3.1.2 SimpleScalar characteristics

We use SimpleScalar [4], version 3.0a, for cycle-true, execution-driven simulations. The architecture of the instruction set of the simulated CPU called Portable ISA (PISA) has been inspired by the MIPS IV instruction set [20]. It uses a 64 bit encoded instruction format while the CPU's internal data path is still 32 bit wide. There are separate reorder buffers for compute and load/store instructions and register renaming facilities to make out-of-order instruction execution possible. A configurable number of execution units can make use of 32 floating point and another set of 32 integer registers. The registers are 32 bit wide. Six different configurable predictors are offered. All stages of the pipeline can be reconfigured. The instruction flow through the pipeline stages is sketched in Fig. 4. The pipeline consists of six stages and thus fits between the five stage pipeline of some Mips R4000 variants and the seven stage pipeline of the Mips R10000. Compared with the pipeline of e.g. an R4300i a distinct issue stage is introduced for dynamic scheduling of instructions. Compared with the R10000 there is only one execution stage. However, the writeback phase is split into two stages in SimpleScalar (writeback and commit stages). The functionality of the different stages can be summarized as follows:

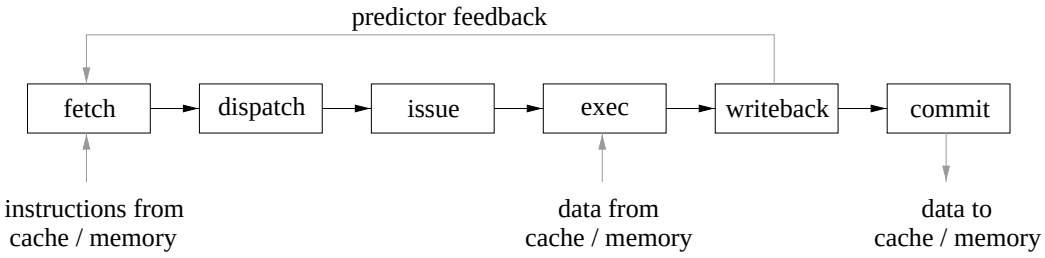

Figure 4: *Instruction flow through the SimpleScalar Pipeline.*

- *fetch:* instructions are fetched from the cache and put into the dispatch queue.
- *dispatch:* instructions from the dispatch queue are decoded and put into the corresponding reorder buffer. Register renaming is applied.
- *issue:* the issue stage determines when operands and functional units are available for the instructions in the reorder buffers and releases instructions accordingly out-of-order to the functional units.
- *exec:* instruction execution stage.
- *writeback:* miss predictions are handled. The results of the execution stage are written back to registers. Moreover, the result is checked whether there are instructions which may be ready for execution now due to resolved dependencies.
- *commit:* store data is commited to the data cache. Instructions are completed in-order.

The cache levels use writeback mode and non-blocking loads and stores. More details can be found in [4].

SimpleScalar has been chosen for the simulations and the implementation of a more detailed main memory system because of the following reasons. The complete source code is freely available (written in C). The simulator implements a cycle-true architecture model of a modern CPU. Each pipeline stage as well as the cache levels can be easily configured by command line options. Since the tool-set already comes with its own cross-compiler, applications can be easily ported to the simulated environment. In comparison to that, SimOS simulates a more complete computing system considering other computer components. However, this complexity was not required in our investigations. Finally, RSIM focuses on shared-memory multiprocessors. It is possible to restrict the simulations to investigations of the architecture of a uniprocessor system. However, in order to familiarize with the source code one has to cope with a lot of mechanisms which are only required for a multiprocessor system but still affect the uniprocessor simulation. For the same reasons, the cross-compilation process is not as straightforward as in SimpleScalar.

SimpleScalar has some restrictions which have not affected our investigations, e.g. the lack of multiprocessor support. Moreover, SimpleScalar only simulates the execution of user-level programs, i.e. some of the needed system calls are handled by the simulation host. Most of them are required for file I/O. However, we have restricted our study to embedded systems without real-time constraints. A lot of processes which are usually performed by the operating system in a PC are executed by specialized modules in an embedded system and thus do not affect the CPU speed. Hence, the kernel of the embedded processor, the CPU, is responsible for the remaining computations and control tasks. There may not even be the need for an operating system.

3.2 Simulator enhancements

The following DRAM and memory controller models have been integrated into SimpleScalar.

3.2.1 DRAM models

SDRAM Using recent 256 Mbit SDRAM chips and a common 64 bit wide memory bus at least 128 MByte of main memory are available. An SDRAM by IBM [10] has been modeled in the 16 bit configuration. Hence, four devices have to be controlled in parallel to form the 64 bit wide bus. This setting is equivalent to a 64 bit DIMM with four mounted devices. The controller sees memory banks four times the size of a single chip's bank. The memory bus is clocked at 100 MHz as in current personal computer systems. The IBM chip consists of four banks with 8192 rows each. The following timing parameters have been modeled, see Fig. 5:

- t_{AA} column address strobe delay: time between issuing a read instruction on the control pins and the appearance of the first data item on the output pins.
- t_{RP} precharge time: time needed to precharge an active row.
- t_{RCD} row address strobe to column address strobe delay: time needed to activate a row of an idle memory bank.
- t_{DPL} data input to precharge delay: time needed between the transfer of the last data item of a burst write and a following precharge operation within the same bank.

- t_{WAR} write after read delay: this parameter considers the bus turn around time to avoid data contention if the data flow direction changes from read to write.

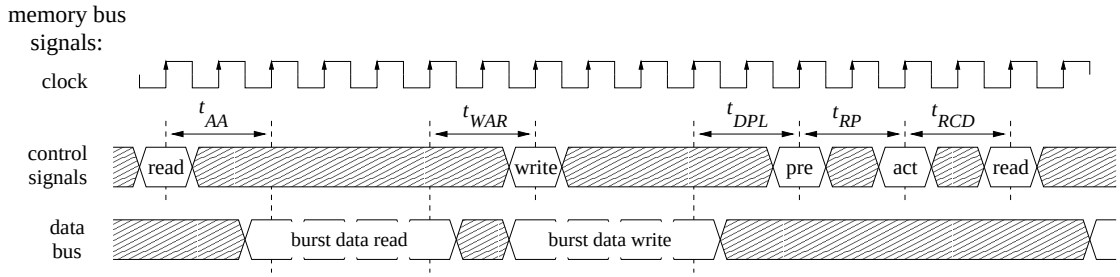

Figure 5: *SDRAM: row hit followed by a row miss in the same memory bank.*

Note that the minimal row active period t_{RAS} cannot be violated with our simulator settings. The CPU uses a cache line size of 32 Byte which corresponds to burst transfers of length four over the memory bus. This burst mode is used by common memory controllers.

Refresh operations have been neglected. The refresh of a single row (an auto refresh cycle) would need seven clock cycles ($70ns$). 8192 auto refresh cycles are needed every $64ms$. Thus, refresh operations take $\frac{8192 \cdot 70ns}{64ms} \cdot 100\% = 0.9\%$ of the run-time of the SDRAM. A refresh of the whole DIMM needs $575\mu s$ and may considerably delay the execution of a CPU instruction in the worst-case. However, the refresh may be split into smaller intervals refreshing only some of the rows at a time.

RDRAM The RDRAM memory bus is 16 bit wide as is an RDRAM chip. Again, 256 Mbit devices are modeled. Thus, the Rambus channel consists of four RDRAMs. The channel configuration allows to control the internal banks of each memory chip individually. The chosen RDRAM [22] consists of 32 banks with 512 rows each. Adjacent banks have to share sense amplifiers since a sense amplifier row only carries 1 KByte of information. A memory row however stores 2 KByte. This is why adjacent banks cannot be in an activated state at the same time. The Rambus channel runs at a 400 MHz clock using both edges of the clock signal and a bus width of 16 bit. The following timing parameters have been modeled, see Fig. 6:

- t_{RCD} row address strobe to column address strobe delay
- t_{CAC} column address strobe access delay
- t_{CWD} column address strobe write delay: time needed from the end of the control packet which initiated the write access to the beginning of the corresponding data packet
- t_{RP} row precharge time
- t_{RTR} retire delay: time needed for data to be transferred from the write buffer of the device to the corresponding sense amplifiers. This task is performed in the background but must be taken into consideration to maintain data consistency if read accesses appear immediately after write accesses.

- t_{RDP} read to precharge delay: time needed from the end of a control packet initiating a read access to the end of a precharge control packet
- t_{PP} precharge to precharge delay for precharge control packets in the same device (not displayed)
- t_{packet} time to transmit a packet

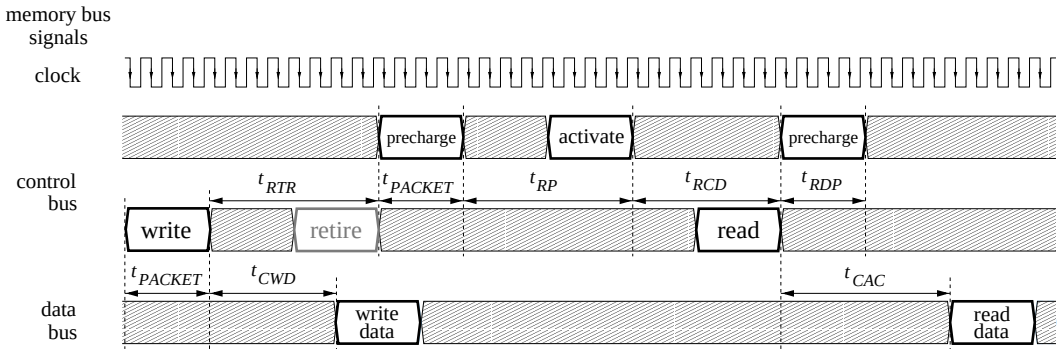

Figure 6: *RDRAM: row hit followed by a row miss in the same memory bank.*

Note that the minimal row active period t_{RAS} of 20 clock cycles can be violated with our simulator settings although the CPU uses a cache line size of 32 Byte which corresponds to two data packets transferred over the Rambus channel. A violation occurs if a row is activated, a read is initiated by a read control packet, and the row is immediately precharged. With our settings, the row active time turns out to be only (using two data and control packets respectively for the transfer) $t_{RCD} + t_{RDP} + t_{packet} = 15 cycles$. The minimal row active period is likewise violated in Fig. 6 looking at the read command. The same control sequence using two write control packets instead of the read control ones however does not violate t_{RAS} because the retire timing must be considered. The descibed violation of t_{RAS} is only possible if the memory controller uses an open-page policy. The closed-page policy timing however does not violate t_{RAS}.

Again, refresh operations have been neglected. Refresh commands can be broadcast through the channel so that the same row of the same bank can be refreshed in all devices at the same time. The refresh of a single row has a latency of $70ns$. However, row refreshes can be performed in an interleaved way using non-adjacent bank numbers. Thus, a single row refresh only occupies $20ns$ on the control signal lines of the memory bus. A RDRAM must be refreshed every $32ms$ and consists of 32 banks with 512 rows each. Therefore, refresh operations take $\frac{512 \cdot 32 \cdot 20ns}{32ms} \cdot 100\% = 1.02\%$ of the run-time of the Rambus channel. Again, the delay of $328\mu s$ introduced by a refresh of a whole device may be reduced by splitting the operation into refreshes of only some of the rows in smaller periods.

Comparing the settings for the DRAMs RDRAMs have been given preference over SDRAMs. The SDRAM system only uses a single DIMM. Several DIMMs however may be accessed with an interleaved scheme.

3.2.2 Memory controller model

The memory controller is responsible for the mapping of physical addresses output by a memory management unit to chip, row, and column addresses in a main memory system.

The virtual memory space of SimpleScalar is shown in Fig. 7. Addresses of the size 32 bit allow a memory space of four GByte. Only the lower two GByte are actually used. The first four MBytes are unused. The code segment of the simulated application may cover up to 252 MByte. The heap segment grows from the 256 MByte boundary (heap base) to higher addresses. The last 16 KByte of the virtual memory space are also unused. Finally, the stack grows from address $0x7fffc000$ corresponding to $2GByte - 16KByte$ (stack base) to lower addresses.

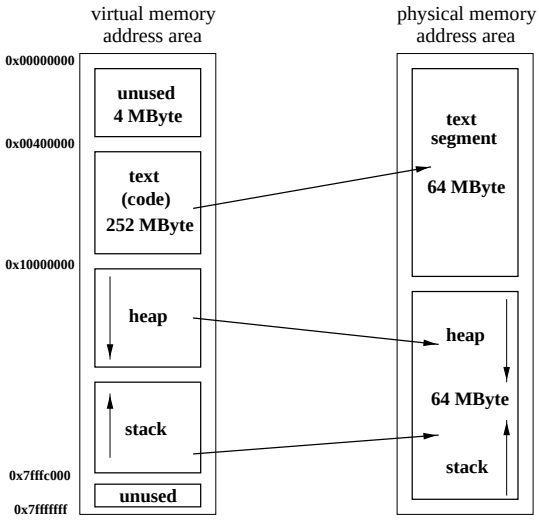

Figure 7: *Virtual to physical memory address mapping.*

The virtual memory space of 2 GByte is now translated to a physical memory space of 128 MByte which is usually done by a memory management unit. This has been implemented as shown in Fig. 7. The code segment is mapped into the first 64 MByte of the physical memory. The upper 64 MByte may be occupied by the heap and stack. The memory controller now has the option to map the physical addresses to chip, row, and column addresses of the main memory. Two variants have been implemented.

Linear translation The memory controller maps physical addresses in a linear manner to the chips and banks of the main memory. That is, passing the boundaries of a DRAM row results in an access of the next row in the same internal DRAM bank (see Fig. 8). The code segment is mapped to the first two banks of the SDRAM DIMM and to the first 64 banks (corresponding to the first two devices) of the RDRAM channel respectively. The base of the heap is mapped to the third bank of the SDRAM DIMM and to the first bank of the third device of the RDRAM channel respectively. Similarly, the base of the stack is mapped to the end of the fourth bank of the SDRAM DIMM and to the end of the 32nd bank of the fourth device of the RDRAM channel.

Interleaved translation In this mode, addresses are mapped in an interleaved manner to the internal banks of the DRAMs in order to take advantage of the bankwise organization. This is a special case of address permutation which is often realized by exchanging some address lines. Whenever the address passes the boundaries of a DRAM row, the next address accesses another bank and not the next row of the same bank. We can make

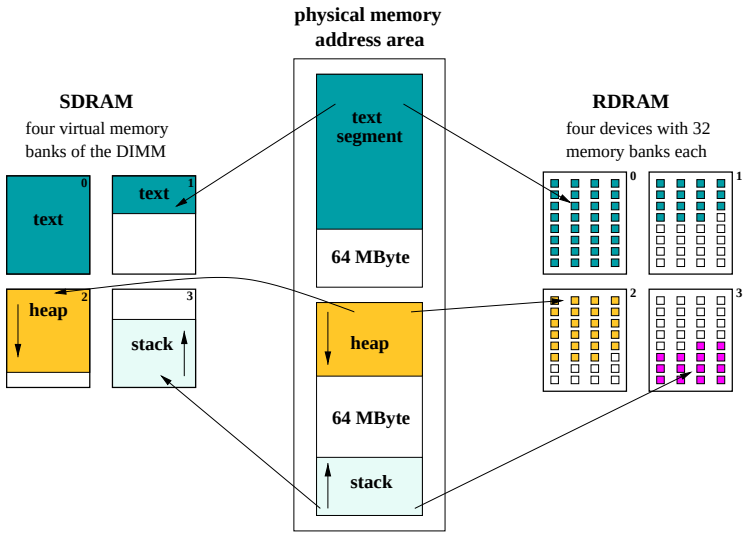

Figure 8: *Linear memory mapping.*

full use of this mode if an application accesses streamed data since changing to another memory bank is usually faster than accessing another memory row in the same memory bank.

The resulting memory mapping can be seen in Fig. 9. The row size for the SDRAM DIMM is 4 KByte with a total of 2^{15} rows spread over four banks of the DIMM. This setting may keep up to 16 KByte of continuous memory opened at the same time.

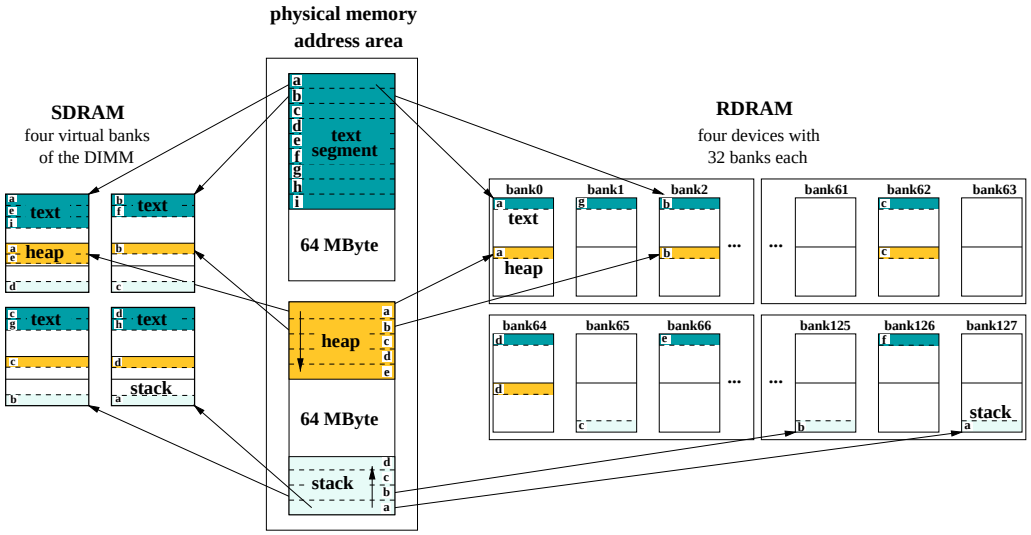

Figure 9: *Interleaved memory mapping.*

Since the modeled RDRAM uses shared sense amplifiers, the next bank is skipped if a row change appears and thus every second bank is allocated consecutively for streamed data. Starting with bank 0 even bank numbers are accessed in steps of two. Then, the odd bank numbers are used. The row size of the RDRAM is 2 KByte with a total of 2^{16}

rows spread over 128 banks in four devices. This scheme may keep up to 128 KByte of continuous memory opened at the same time.

Page activation policies Usually, an *open-page* policy and a *closed-page* policy are distinguished. Using an open-page scheme a memory row is kept activated as long as possible. This way, accesses bounded to the current row do not need to precharge and activate the row again. However, row changes may take longer since the activated row must be precharged first. If row changes are likely to happen frequently, applying the closed-page policy may result in better performance. In closed-page mode the memory controller precharges an active row as soon as possible.

Overlapped processing Since the controller is connected to two independent buses, it is able to process an access on the memory bus while receiving further requests through the front-side bus (or an internal bus if the controller is integrated in the embedded CPU). Note that the CPU must support this mode of operation by non-blocking load/store execution. Thus, the controller may buffer requests and reschedule them in order to hide some of the latency introduced by activation and precharge tasks taking advantage of the pipelined interface of recent DRAMs. An example is given in Fig. 10 using only a single buffer and open-page mode. The first operation is a write to the current memory bank.

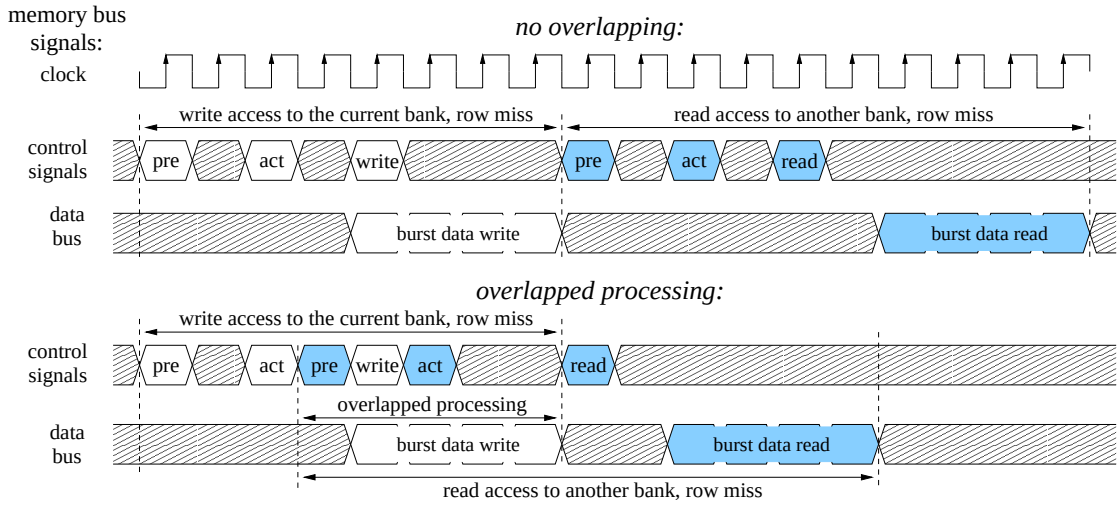

Figure 10: *Overlapped processing, SDRAM case.*

However, due to a row miss another row must be opened. This is why a precharge and an activation are needed. The same is true for the second operation which is a read access to another bank of the memory. Again, a row miss occurs leading to additional precharge and activation commands. Now assume that the second operation, the read access, is known during the processing of the first access, since the corresponding request has already been transferred through the front-side bus. The latency for precharging and activation can be completely hidden. The amount of clock cycles which can be overlapped depends on the current as well as the buffered (next) access and the inter-arrival time of requests at the controller. In this example, overlapped processing saves four clock cycles.

The implemented memory controller has the following operation modes. Linear or interleaved address translation can be chosen. In addition, overlapped processing is sup-

ported in open-page mode. However, in order to keep the controller simple the controller only buffers one further request. Finally, the closed-page policy can be set. Overlapped processing is not supported in closed-page mode. Therefore, linear and interleaved address translation are not distinguished in closed-page mode since both translation schemes result in the same latencies. In closed-page mode, only two different cases must be distinguished for SDRAMs and RDRAMs respectively because only the request type (read or write) is important. Using open-page mode without overlapped processing and SDRAMs 18 different cases are determined depending on the request type of the current and the buffered operation, the bank addresses, and the row state (row hit/miss) of both operations. For RDRAMs, 36 cases have been identified. The higher number of cases compared to the SDRAM controller is mainly caused by the shared sense amplifiers of the modeled RDRAM, the write buffer, and the use of several concurrent devices. Up to two adjacent memory banks must be precharged if a random memory bank is accessed. If the overlapped processing mode is enabled in addition, the overlapping period may be of variable size. The implementation of the memory controller is described in detail in appendix A.

4 Experimental results

4.1 Applications

Most of the programs used for simulation can be found in [2], except *tmn* and *gzip*. We do not only look at typical CPU benchmarks but also investigate real-world programs. The programs can be classified into three areas:

CPU benchmarks

- *Dhrystone 2.1:* this benchmark looks at the integer performance of a processor and provides a MIPS rating based on a typical instruction mix.
- *Linpack:* measures the floating-point performance of a system based on linear algebra routines.
- *Whetstones:* another floating-point benchmark based on a typical instruction mix.

Computer Science benchmarks

- *Matrix multiply (mm) 1.0:* nine different algorithms for computing matrix multiplications can be performed. We have restricted the matrix dimensions to 300×300 elements and used an unoptimized algorithm.
- *Fhourstones (c4) 1.0:* integer-only program implementing a connect-4 game solver. It performs an exhaustive search with the help of large hash table structures. More than one million 32 bit hash entries are required.
- *Sieve of Eratosthenes (nsieve) 1.2b:* an integer program which computes prime numbers based on the algorithm of Eratosthenes. Arrays of several MBytes are used.
- *Shuffle:* the program shuffles four decks of cards using a random number generator for 26000 times.
- *Heapsort 1.0:* sorts a random array with up to 2^{19} integer elements.

Applications

- *fft (tfft):* this program computes a fast Fourier transform for input signal sizes from 16 to 2^{16} points.
- *h.263 video decoder (tmn) 3.1.2:* implementation by the University of British Columbia and Telenor Research of the ITU-T recommendation H.263 *Video coding for low bit rate communication*. 100 frames of the test sequence *Miss Amerika* in the qcif picture format are decoded. The sequence has been encoded with an average data rate of 1 kbit per frame.
- *DNA segment compare (sim):* finds the seven best non-intersecting alignments between two DNA segments of 2500 elements using dynamic programming techniques.
- *gzip 1.2.4a:* compression program using Lempel-Ziv coding. Gzip compresses its own distribution archive (approx. 2.5 Mbyte in size).

4.2 Simulations

4.2.1 Influence of the memory controller

In order to compare SDRAMs with RDRAMs under different memory controller configurations in an embedded system scenario, the following settings for SimpleScalar have been chosen. All simulations have been performed on Sun Ultra workstations (big-endian architecture). The CPU runs at a clock speed of 200 MHz and is two-way superscalar with out-of-order issue. Bimodal prediction with 512 entries is used. The reorder buffers have eight entries for compute and four entries for load/store instructions. The direct mapped first level cache is split into an 8 KByte instruction cache and an 8 KByte data cache. The cache line size is 32 Byte. There is no second level cache. Loads and stores are non-blocking. However, the memory controller limits the number of outstanding main memory accesses. The following functional units are available: an integer ALU, an integer multiplier/divider, a floating-point ALU, and a floating-point multiplier/divider.

As described in section 3.2.2, the memory controller can apply linear (*lin*) as well as interleaved (*int*) address translation. Moreover, a closed-page (*c-p*), an open-page (*o-p*), and an open-page activation policy with overlapped processing can be chosen. Finally, SDRAMs and RDRAMs can be controlled. The controller is supposed to be integrated in the embedded CPU. At most two main memory accesses can be in a pending state.

The simulated execution times of the different programs are displayed in Fig. 11 and Tab. 2 respectively. The values have been scaled to the execution time under the SDRAM in closed-page policy configuration. Tab. 3 provides information about program execution statistics: cache miss rates, the amount of main memory accesses scaled to the number of all executed instructions (*total*), the ratio between the number of memory instructions that can make use of overlapped processing to the number of all main memory accesses, the amount of loads and stores scaled to the number of executed instructions (*rel.*), and the absolute count of executed load and store instructions. Since the amount of memory requests which can make use of overlapped processing does not vary remarkably by changing the address translation mode or the DRAM type, only the maximum of the corresponding simulation runs is considered in the table. Finally, Tab. 4 shows hit and miss statistics for the main memory system using open-page mode. Looking at two consecutive memory requests, the next pending request may access the same row as the current access. This

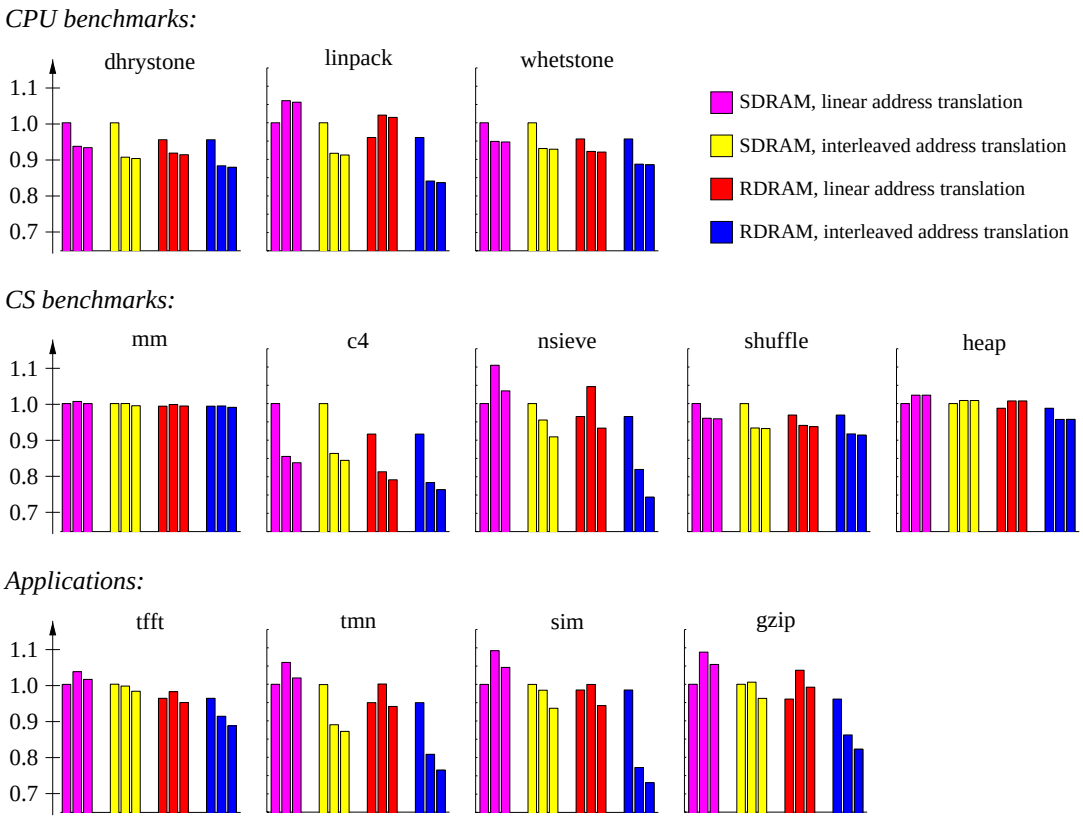

Figure 11: *Comparison of the simulated execution times of various programs using different memory controller implementations: closed-page policy (left bar), open-page policy (middle bar), and open-page policy with overlapped processing (right bar). The execution times have been scaled to the execution time using SDRAMs combined with the closed-page policy (leftmost bar).*

situation is called a hit in the current row (*cr hit*). If the pending request wants to access a different row in the same bank, a miss in the current row happens (*cr miss*). Finally, the situations where the pending request will hit or miss a row in another bank than the current one are summarized as accesses to another bank (*ob*). The negligible amount of accesses to idle banks is not quoted in the table.

For two programs the execution time is almost independent on the memory and controller used: matrix multiply (*mm*) and *heapsort*. Both programs show almost no misses in the instruction cache and moderate miss rates in the data cache: 6.6% for *mm*, 2.7% for *heapsort*. Although load and store instructions take around 20% of the executed instructions, only slightly more than 1% of the executed instructions result in a main memory access. This rate is too low to produce any run-time effects. For instance, looking at SDRAMs and the *mm* program, the rate of expensive row misses in the current bank can be reduced from 76% for linear address translation to 21% using interleaved address translation. However, there is no noticeable difference in the execution time.

Another group of programs, *whetstone, shuffle, dhrystone*, and *c4*, runs faster by controlling RDRAMs instead of SDRAMs. Moreover, using interleaved address translation also always shortens the execution time. In most configurations however the differences

program	*addr. transl.*	*SDRAM* c-p	o-p	overl.	*RDRAM* c-p	o-p	overl.
mm	lin	1.0	1.01	1.0	0.99	1.00	0.99
	int		1.00	0.99		0.99	0.99
heapsort	lin	1.0	1.02	1.02	0.99	1.01	1.01
	int		1.01	1.01		0.96	0.96
whetstone	lin	1.0	0.95	0.95	0.96	0.92	0.92
	int		0.93	0.93		0.89	0.89
shuffle	lin	1.0	0.96	0.96	0.97	0.94	0.94
	int		0.93	0.93		0.92	0.91
dhrystone	lin	1.0	0.94	0.93	0.95	0.92	0.91
	int		0.91	0.90		0.88	0.88
c4	lin	1.0	0.86	0.84	0.92	0.81	0.79
	int		0.86	0.84		0.78	0.76
nsieve	lin	1.0	1.10	1.03	0.96	1.05	0.93
	int		0.96	0.91		0.82	0.74
sim	lin	1.0	1.09	1.05	0.98	1.00	0.94
	int		0.98	0.93		0.77	0.73
tmn	lin	1.0	1.06	1.02	0.95	1.00	0.94
	int		0.89	0.87		0.81	0.76
tfft	lin	1.0	1.03	1.01	0.96	0.98	0.95
	int		1.00	0.98		0.91	0.89
gzip	lin	1.0	1.09	1.05	0.96	1.04	0.99
	int		1.01	0.96		0.86	0.82
linpack	lin	1.0	1.06	1.06	0.96	1.02	1.01
	int		0.92	0.91		0.84	0.84

Table 2: *Simulated program execution times for different memory controller schemes.*

program	*L1 cache miss [%]* instr.	data	*main mem. accesses* total [%]	overl. [%]	*exec. load/store* rel. [%]	abs. [10^6]
mm	0.0	6.6	1.2	17.0	19.2	409
heapsort	0.0	2.7	1.1	0.3	22.1	310
whetstone	3.5	0.0	3.7	0.3	32.5	12
shuffle	2.0	0.6	2.2	4.5	21.1	196
dhrystone	3.3	0.0	3.5	2.6	40.9	443
c4	10.9	3.4	13.8	8.4	32.6	1229
nsieve	0.0	25.9	5.9	45.0	11.7	113
sim	2.3	17.9	11.4	18.2	36.3	1595
tmn	1.2	8.5	7.1	27.3	55.5	250
tfft	1.0	7.7	5.5	15.7	38.9	80
gzip	0.0	17.0	5.3	28.6	28.0	138
linpack	6.1	8.9	4.5	3.7	26.9	21

Table 3: *Program execution statistics.*

are negligible. Applying overlapped processing does not improve the performance noticeablely. Using open-page mode instead of closed-page mode improves the execution time by at least 3% (*shuffle*, RDRAM, linear translation), and up to at most 15% (*c4*, RDRAM, interleaved translation). The programs have in common that they generate moderate miss rates in the instruction cache and almost no misses in the data cache, see Tab. 3. Since the share of loads and stores of all executed instructions is high (up to 40.9% for *dhrystone*), a noticeable amount of the simulated instructions result in main memory accesses. Moreover, the accesses to main memory are very localized. At least 68.5% of the memory accesses hit in the same memory row as the preceding memory access. Thus,

program	*addr. transl.*	*SDRAM [%]* cr hit	cr miss	ob	*RDRAM [%]* cr hit	cr miss	ob
mm	lin	16.2	76.0	7.8	1.5	39.4	59.1
	int		21.1	62.7		1.0	97.5
heapsort	lin	4.4	91.7	3.9	3.1	59.5	37.4
	int		32.2	63.4		0.8	96.1
whetstone	lin	77.9	21.9	0.2	77.9	21.9	0.2
	int		8.9	13.2		0.0	22.1
shuffle	lin	68.5	18.8	12.7	68.5	18.8	12.7
	int		0.0	31.5		0.0	31.5
dhrystone	lin	84.2	15.8	0.0	78.9	21.1	0.0
	int		0.0	15.8		0.0	21.1
c4	lin	79.5	8.9	11.6	77.3	9.5	13.2
	int		7.5	13.0		1.3	21.4
nsieve	lin	1.6	98.3	0.0	1.4	88.4	10.2
	int		24.9	73.5		0.3	98.3
sim	lin	30.1	59.1	10.8	30.1	59.1	10.8
	int		9.7	60.2		0.1	69.8
tmn	lin	16.1	74.0	9.9	15.8	70.9	13.3
	int		9.8	74.1		0.2	84.0
tfft	lin	29.3	62.5	8.2	28.3	63.1	8.6
	int		42.0	28.7		21.7	49.9
gzip	lin	7.6	92.3	0.1	6.5	93.4	0.1
	int		25.5	66.9		0.1	93.4
linpack	lin	9.6	88.0	2.4	8.8	88.9	2.3
	int		15.2	75.2		0.1	91.1

cr hit/miss: hit/miss in the current row *ob:* access to another bank

Table 4: *Main memory access statistics in percent of all memory accesses.*

the memory controller benefits well from the open-page policy. The remaining amount of accesses however, which miss in the current bank or access other banks than the one of the preceding access, is too small to influence system performance by changing from linear to interleaved memory translation. Moreover, only 8.4% of all memory accesses can make use of overlapped processing in the best case.

The next group of programs, consisting of *nsieve, sim, tmn*, and *tfft*, shows a noticable performance improvement by using overlapped proccessing. From 15.7% (*tfft*, RDRAM, interleaved translation) up to 45.0% (*nsieve*, SDRAM, linear translation) of all main memory accesses can make use of this mode. Thus, the execution time can be shortened up to 11% using overlapped processing (*nsieve*, RDRAM, linear translation). Using interleaved address translation, the applications always perform better than using linear translation. Interleaved translation shortens the execution time by up to 23% (*sim*, RDRAM, without overlapped processing). Interestingly, in open-page mode the programs *nsieve* and *tmn* run faster using SDRAMs and interleaved translation compared with RDRAMs and linear translation. All these effects are noticeable since the programs generate high miss rates in the data cache, see Tab. 3. Combined with a high count of executed load and store instructions, at least 5.5% of all executed instructions result in a main memory access. The significant performance improvements by using interleaved address translation instead of linear translation can be achieved because at least 59.1% of all memory accesses cause a row miss in the same memory bank which has been accessed by the preceding access. Using interleaved translation, these relatively expensive accesses can be exchanged for cheaper accesses in other memory banks. For instance, the high miss rate in the current bank

by the *nsieve* program using SDRAMs can be reduced from 98.3% to 24.9%. The high miss rate in the current bank also explains why the DRAM configurations using open-page mode and linear translation perform poorer than the corresponding configurations using closed-page mode. A closed-page policy is able to hide some of the precharge overhead which may be caused by frequent memory row changes.

The remaining programs, *gzip* and *linpack*, behave similar to the preceding group. Since both programs generate very high miss rates in the current memory bank, neither SDRAMs nor RDRAMs can benefit from open-page mode if linear address translation is applied. Thus, interleaved address translation is able to produce much better results since both programs generate high miss rates in the current bank using linear mode. Finally, the difference in execution time applying overlapped processing is only visible looking at *gzip*. Around 28% of all main memory accesses generated by *gzip* can be processed partially concurrently, resulting in around 5% shorter execution times. In the case of *linpack*, however, the amount of overlapping requests is too small to show any improvements in the execution time.

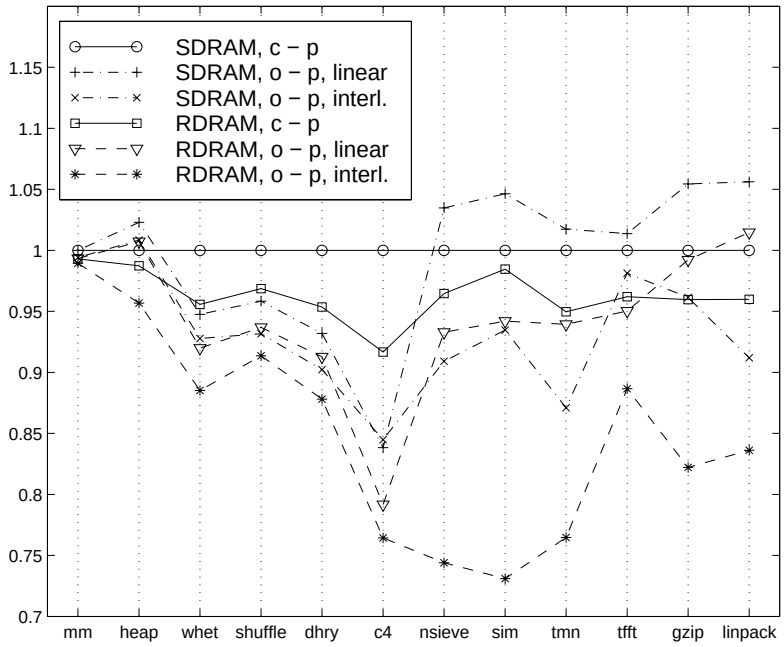

Figure 12: *Normalized simulated execution times for closed-page and open-page modes with overlapped processing. SDRAM with closed-page mode is used as reference.*

The results are summarized in Fig. 12. On the one hand, the RDRAM configuration using open-page mode and interleaved address translation always outperforms all other configurations. The performance gain compared with all other configurations ranges from some percent to up to 21% for the *sim* application. On the other hand, the RDRAM more heavily depends on the chosen memory controller access scheme. In the worst case, the *sim* program looses more than 34% of its best-case performance by using RDRAMs and closed-page mode. SDRAMs using open-page mode and interleaved address translation can well compete with RDRAMs using linear translation. The performance of the SDRAM is not that dependent on the memory controller access scheme as RDRAM's performance is. Thus, RDRAMs should be controlled with an open-page access strategy. Finally, using

interleaved address translation mode always seems to be a good choice for both DRAM types.

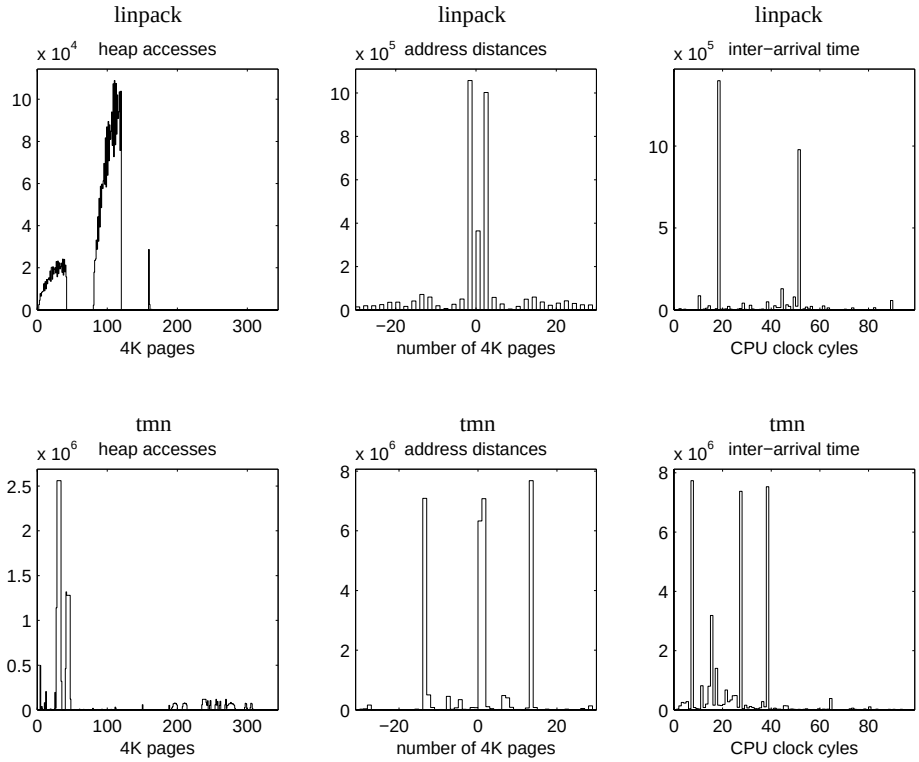

Figure 13: *Access histograms for* linpack *and* tmn.

Analysis of traces The conclusions from our simulations can be explained by an analysis of access histograms which point out the locality properties of the applications. In Fig. 13, the results for two applications, *linpack* and *tmn*, are displayed. The left graphs show the frequencies of accesses in the heap segments. Since 95.9% (*linpack*) and 79.4% (*tmn*), respectively, of all main memory requests access the heap, only this memory area is displayed. The middle graphs show the frequency of address distances of consecutive memory accesses. The right graphs illustrate the inter-arrival times of consecutive memory requests seen by the memory controller in CPU clock cycles. All diagrams use a resolution of a virtual SDRAM DIMM page (4 KByte) and the corresponding traces have been recorded in SimpleScalar for a memory controller using linear address translation, overlapped processing, and SDRAMs.

Both programs have shown weak hit rates in the current bank. On the one hand, *linpack* was not able to make use of overlapped processing. On the other hand, *tmn* showed a noticeable increase in performance by using overlapped processing. The reason for this behavior is displayed in the right graphs of Fig. 13. The inter-arrival time histogram for *linpack* shows two peaks at 18 and 51 clock cycles which already cover more than 68% of all accesses. However, SDRAM accesses hardly ever need more than 20 CPU clock cycles to finish in our configuration. Therefore, the amount of overlap between consecutive main memory requests is too small to be able to take advantage of it. The situation however is different for *tmn*. Its inter-arrival diagram shows peaks at seven and around 15 clock

cycles. Memory requests appear at higher densities so that the memory controller can make use of overlapped processing.

The remaining diagrams in Fig. 13 point out that the size of an SDRAM row is too small to exploit the inherent locality of the applications. More than 90% of all main memory requests of *linpack* access two data regions of the size 160 KByte each. Address changes plus/minus two pages appear often. Therefore, using linear address translation, memory row exchanges must often be performed due to high miss rates in the current active row. This statement is also true for *tmn*. Two memory regions of the size 32 KByte each are accessed most often. Moreover, accesses seem to alternate between these two regions since address changes of plus/minus 13 pages appear frequently which is also the mean distance between the two regions.

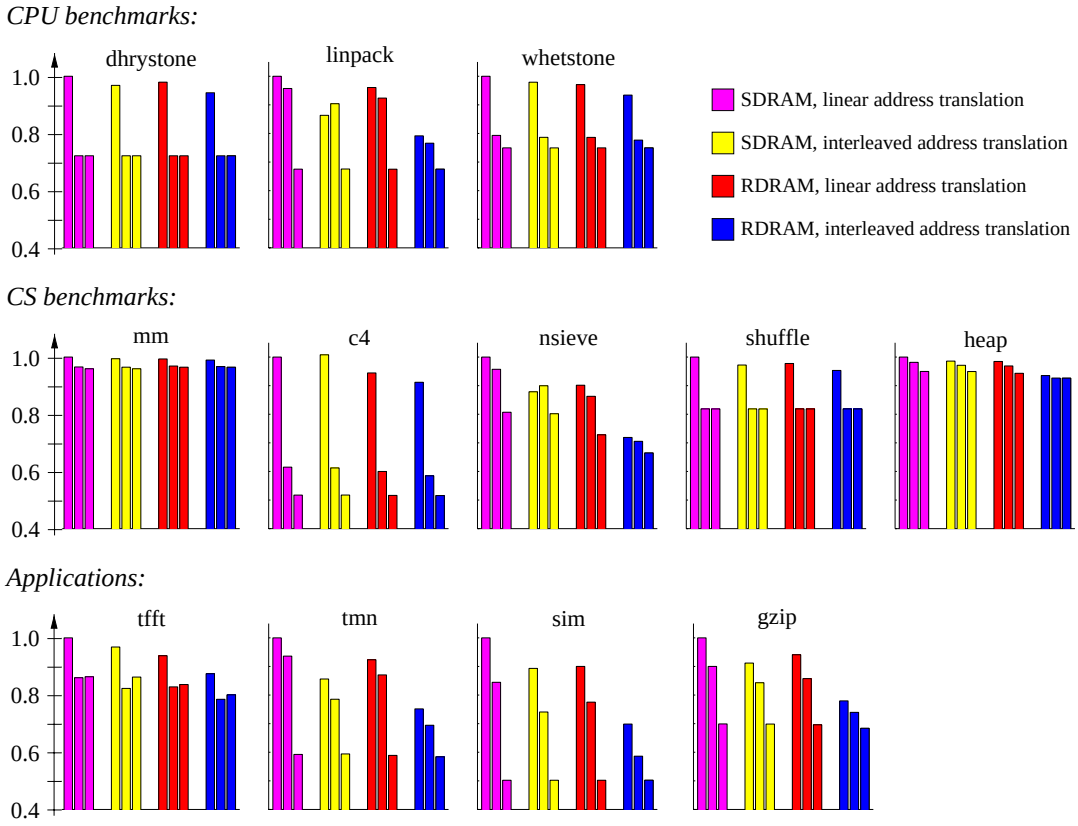

Figure 14: *Comparison of the simulated execution times of various programs using different cache configurations: original 8 Kbyte + 8 Kbyte first level cache (left bar), doubled first level cache (middle bar), and an additional second level cache of 128 Kbyte (right bar). The execution times have been scaled to the execution time using SDRAMs combined with the original 8 Kbyte + 8 Kbyte cache configuration (leftmost bar).*

4.2.2 Influence of the cache size

In order to complete our experiences with the benchmarks and the CPU settings we have varied the cache size of the CPU. The following simulations have been performed using an open-page strategy with linear as well as interleaved address translation, overlapped mode

enabled. In the first experiment, we have doubled the size of the first level cache. Thus, the first level cache is split in a 16 Kbyte two-way set associative instruction cache as well as a data cache of the same size and organization. In the second experiment, an unified second level cache of 128 Kbyte, four-way set associative, has been added to the doubled first level cache running at CPU clock speed (200 MHz). The second level cache also uses a line size of 32 Byte. The resulting execution times of the programs are displayed in Fig. 14 and Tab. 5 respectively.

program	*addr. transl.*	*SDRAM*			*RDRAM*		
		L1 8 + 8	L1 16 + 16	L1 + L2	L1 8 + 8	L1 16 + 16	L1 + L2
mm	lin	1.0	0.97	0.96	0.99	0.97	0.96
	int	0.99	0.96	0.96	0.99	0.97	0.96
heapsort	lin	1.0	0.98	0.95	0.98	0.97	0.94
	int	0.99	0.97	0.95	0.94	0.93	0.93
whetstone	lin	1.0	0.79	0.75	0.97	0.79	0.75
	int	0.98	0.79	0.75	0.93	0.78	0.75
shuffle	lin	1.0	0.82	0.82	0.98	0.82	0.82
	int	0.97	0.82	0.82	0.95	0.82	0.82
dhrystone	lin	1.0	0.72	0.72	0.98	0.72	0.72
	int	0.97	0.72	0.72	0.94	0.72	0.72
c4	lin	1.0	0.62	0.52	0.94	0.60	0.52
	int	1.01	0.61	0.52	0.91	0.58	0.52
nsieve	lin	1.0	0.96	0.81	0.90	0.86	0.73
	int	0.88	0.90	0.80	0.72	0.71	0.66
sim	lin	1.0	0.84	0.50	0.90	0.78	0.50
	int	0.89	0.74	0.50	0.70	0.59	0.50
tmn	lin	1.0	0.94	0.59	0.92	0.87	0.59
	int	0.86	0.79	0.59	0.75	0.69	0.58
tfft	lin	1.0	0.86	0.86	0.94	0.83	0.84
	int	0.97	0.82	0.86	0.87	0.79	0.80
gzip	lin	1.0	0.90	0.70	0.94	0.86	0.70
	int	0.91	0.84	0.70	0.78	0.74	0.68
linpack	lin	1.0	0.96	0.68	0.96	0.92	0.68
	int	0.86	0.90	0.68	0.79	0.77	0.68

Table 5: *Simulated program execution times for different cache configurations.*

As stated in the preceding section, the programs *mm* and *heapsort* already fit well in the original cache size. Thus, the additional cache does not accelerate these programs any further. The programs *whetstone, dhrystone, shuffle*, and *tfft* generate considerable miss rates in the original configuration. However, with a doubled first level cache, the miss rates decrease already to a level where the underlying main memory system no longer influences the execution time noticeably. Most of the remaining programs show this independence of the main memory if the second level cache is added. In fact, looking at *tfft* the management of the additional second level cache may even slow down the system. Only *nsieve* still shows different execution times dependent on the memory type and the used memory controller address translation caused by a miss rate of over 30% in the second level cache.

4.2.3 Influence of functional units and the clock frequency

We have performed additional simulations to investigate the transition from embedded systems to more general-purpose systems by raising the clock frequency of the CPU and by adding more superscalar pipelines, functional units, and reorder buffer entries. The

CPU is now four-way superscalar with out-of-order issue. Bimodal prediction with 2048 entries (four times the original size) is used. The reorder buffers have 32 entries for compute and 16 entries for load/store instructions (four times the original number). The first level cache size has been kept with an 8 KByte instruction and an 8 KByte data cache, There is no second level cache. The number of functional units has been doubled: two integer ALUs, two integer multipliers/dividers, two floating-point ALUs, and two floating-point multipliers/dividers. The simulated execution times of the applications are visualized in Fig. 15 and Tab. 6 respectively.

program	*addr. transl.*	*SDRAM*			2×clk	*RDRAM*			2×clk
		orig.	4×way	2×clk	4×way	orig.	4×way	2×clk	4×way
mm	lin	1.0	0.56	0.56	0.34	0.99	0.55	0.53	0.31
	int	0.99	0.55	0.55	0.33	0.99	0.55	0.53	0.31
heapsort	lin	1.0	0.61	0.58	0.38	0.98	0.59	0.56	0.37
	int	0.99	0.59	0.57	0.37	0.94	0.54	0.51	0.31
whetstone	lin	1.0	0.70	0.66	0.51	0.97	0.67	0.63	0.47
	int	0.98	0.68	0.64	0.49	0.93	0.63	0.59	0.44
shuffle	lin	1.0	0.57	0.61	0.40	0.98	0.55	0.59	0.37
	int	0.97	0.55	0.59	0.37	0.95	0.53	0.56	0.34
dhrystone	lin	1.0	0.68	0.66	0.50	0.98	0.67	0.63	0.47
	int	0.97	0.65	0.63	0.47	0.94	0.63	0.59	0.43
c4	lin	1.0	0.83	0.84	0.75	0.94	0.77	0.76	0.67
	int	1.01	0.84	0.85	0.76	0.91	0.74	0.72	0.63
nsieve	lin	1.0	0.82	0.80	0.73	0.90	0.72	0.69	0.61
	int	0.88	0.69	0.67	0.59	0.72	0.52	0.48	0.39
sim	lin	1.0	0.88	0.88	0.84	0.90	0.78	0.77	0.72
	int	0.89	0.79	0.77	0.74	0.70	0.57	0.56	0.49
tmn	lin	1.0	0.68	0.79	0.60	0.92	0.59	0.70	0.50
	int	0.86	0.54	0.65	0.46	0.75	0.45	0.52	0.35
tfft	lin	1.0	0.74	0.76	0.64	0.94	0.67	0.69	0.56
	int	0.97	0.71	0.73	0.61	0.87	0.61	0.62	0.49
gzip	lin	1.0	0.75	0.75	0.62	0.94	0.69	0.68	0.55
	int	0.91	0.66	0.66	0.52	0.78	0.53	0.49	0.36
linpack	lin	1.0	0.62	0.73	0.54	0.96	0.58	0.69	0.49
	int	0.86	0.49	0.60	0.41	0.79	0.41	0.52	0.32

Table 6: *Simulated program execution times for different CPU configurations.*

The applications which do not heavily depend on the main memory system such as *mm, heapsort*, and *shuffle* can almost completely exploit faster CPUs. All applications can take a noticeable advantage of a doubled clock frequency ($2 \times clk$) or more computing resources ($4 \times way$). For Applications, however, which often have to access the main memory there is no further remarkable gain by using a higher clock frequency together with more computing resources. Thus, the speedup varies from 3.2 (*mm*) down to 1.2 (*sim*). Nevertheless, the relative differences between the DRAM configurations are more severe than in the original configuration without changing the qualitative results stated in section 4.2.1.

5 Conclusion

This report has investigated the influence of recent DRAM technologies, SDRAMs and Direct Rambus RAMs, on embedded systems performance. Especially, the impact of the

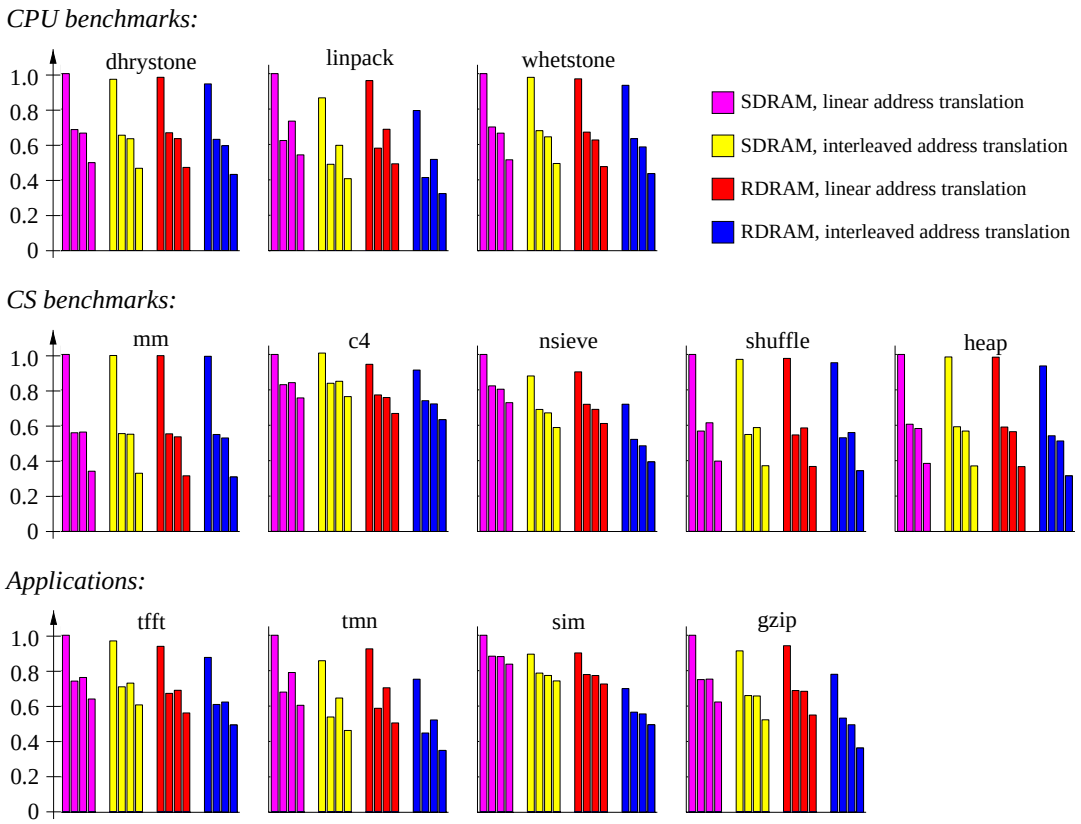

Figure 15: *Comparison of the simulated execution times of various programs using different CPU configurations (from left to right): original CPU (left bar), four-way superscalar (second bar), doubled clock frequency (third bar), and doubled clock frequency and four-way superscalar (right bar). The execution times have been scaled to the execution time using the original CPU with SDRAMs combined with the open-page policy and overlapped processing (leftmost bar).*

memory controller on the resulting performance has been explored by simulating a modern 32 bit out-of-order superscalar CPU architecture together with different main memory access schemes which can be easily implemented in the controller. Since embedded systems use fewer cache levels and smaller cache sizes due to power, area, and cost constraints, their performance particularly depends on the main memory system. The simulation of 12 different applications showed the following results:

- In general, applying an open-page access heuristic results in better system performance than a closed-page access scheme.
- Exploiting the internal multi-bank structure of modern DRAMs by an interleaved address translation scheme accelerates the program execution by up to 23%. This scheme always seems to be a reasonable choice since expensive row misses can be exchanged against cheaper accesses to other banks complying more the locality characteristics of the applications.
- Using an additional request buffer in order to overlap the processing of two successive memory accesses may decrease the execution time by up to 11%.

- RDRAMs outperform SDRAMs by up to 21% using the same settings. However, RDRAMs depend more on the chosen access scheme than SDRAMs.

For example, an SDRAM with overlapped processing of consecutive memory requests and interleaved bank accesses can well compete with an RDRAM which does not use interleaved addressing. Thus, from a performance point of view, the transition from the mature SDRAM to the new RDRAM technology is beneficial only if the memory controller is optimized.

References

[1] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *25th Annual International Symposium on Computer Architecture*, pages 3–14, 1998.

[2] BenchWeb. Benchmark branch of the Netlib repository, University of Tennessee. *http://www.netlib.org/benchweb/*.

[3] Keith Boland and Apostolos Dollas. Predicting and precluding problems with memory latency. *IEEE Micro*, 14(4):59–67, August 1994.

[4] Doug Burger and Todd. M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[5] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *23th International Symposium on Computer Architecture*, pages 78–89, 1996.

[6] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: building a smarter memory controller. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 70–79, 1999.

[7] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A performance comparison of contemporary DRAM architectures. In *26th International Symposium on Computer Architecture*, pages 222–233, 1999.

[8] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *24th International Symposium on Computer Architecture*, pages 133–143, 1997.

[9] Sung I. Hong, Sally A. McKee, Maximo H. Salinas, Robert H. Klenke, James H. Aylor, and Wm. A. Wulf. Access order and effective bandwidth for streams on a Direct Rambus memory. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 80–89, 1999.

[10] IBM Corp. *256Mb Synchronous DRAM, Die Revision A, IBM0325(40,80,16,4B)4*, June 1999.

[11] Intel Corp. *PC SDRAM Registered DIMM Design Support Document*, rev. 1.2 edition, October 1998.

[12] Intel Corp. *PC SDRAM Specification*, rev. 1.63 edition, October 1998.

[13] Intel Corp. *PC SDRAM unbuffered DIMM Specification*, rev. 1.0 edition, February 1998.

[14] Joint Electron Device Engineering Council (JEDEC). *Standard 21C (JESD21C): Configurations for solid state memories: official and preliminary releases.*

[15] Masaki Kumanoya, Toshiyuki Ogawa, Yasuhiro Konishi, Katsumi Dosaka, and Kazuhiro Shimotori. Trends in high-speed DRAM architectures. *IEICE Transactions on Electronics*, E79-C(4):472–481, April 1996.

[16] Peter S. Magnusson, Fredrik Dahlgren, Hakan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner. SimICS/sun4m: A virtual workstation. In *USENIX Annual Technical Conference*, New Orleans, Louisiana, USA, June 1998.

[17] Sally A. McKee, Robert H. Klenke, Kenneth L. Wright, William A. Wulf, Maximo H. Salinas, James H. Aylor, and Alan P. Batson. Smarter memory: improving bandwidth for streamed references. *IEEE Computer*, 31(7):54–63, July 1998.

[18] Yoichi Oshima, Bing J. Sheu, and Steve H. Jen. High-speed memory architectures for multimedia applications. *IEEE Circuits and Devices Magazine*, 13(1):8–13, January 1997.

[19] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *Third Workshop on Computer Architecture Education*, February 1997.

[20] Charles Price. *MIPS IV Instruction Set, revision 3.1.* Mips Technologies Inc., Mountain View, CA, USA, January 1995.

[21] Betty Prince. *High Performance Memories: New Architecture DRAMs and SRAMs - Evolution and Function, revised ed.* John Wiley & Sons Ltd., 1999.

[22] Rambus Inc. *Direct RDRAM 256/288-MBit (512K x16/18 x 32s), advance information*, August 1998.

[23] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Steve Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[24] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *15th ACM Symposium on Operating Systems Principles*, pages 285–298, 1995.

[25] SPARC International, Santa Clara, CA, USA. *The SPARC Architecture Manual, Version 9.*

[26] Standard Performance Evaluation Corporation. Open Systems Group, CPU benchmark suite. *http://www.specbench.org.*

[27] Kenneth M. Wilson and Kunle Olukotun. Designing high bandwidth on-chip caches. In *24th International Symposium on Computer Architecture*, pages 121–132, 1997.

[28] William A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *Computer-Architecture-News*, 23(1):20 – 24, March 1995.

A Implementation

A.1 Memory accesses in SimpleScalar

Memory accesses are generated by the fetch, issue, and commit stages of the instruction pipeline. In the fetch phase instructions are loaded from the code segment of the memory. During the issue phase, load instructions are issued to the memory hierarchy. Finally, store instructions are processed in the commit phase. The sequence of instructions in SimpleScalar for the simulation of a memory access is visualized as a flow graph combined with pseudo-code in Fig. 16. The cache works with a *write-back* mechanism. A write-back is initiated in the `cache_access()` function which again calls one of the access functions. Whenever a load is initiated in the fetch or issue stage, the function `cache_access()` is called that returns the latency of this cache access. In case of a cache miss (first level or second level cache miss), the `cache_access()` function calls one of the functions

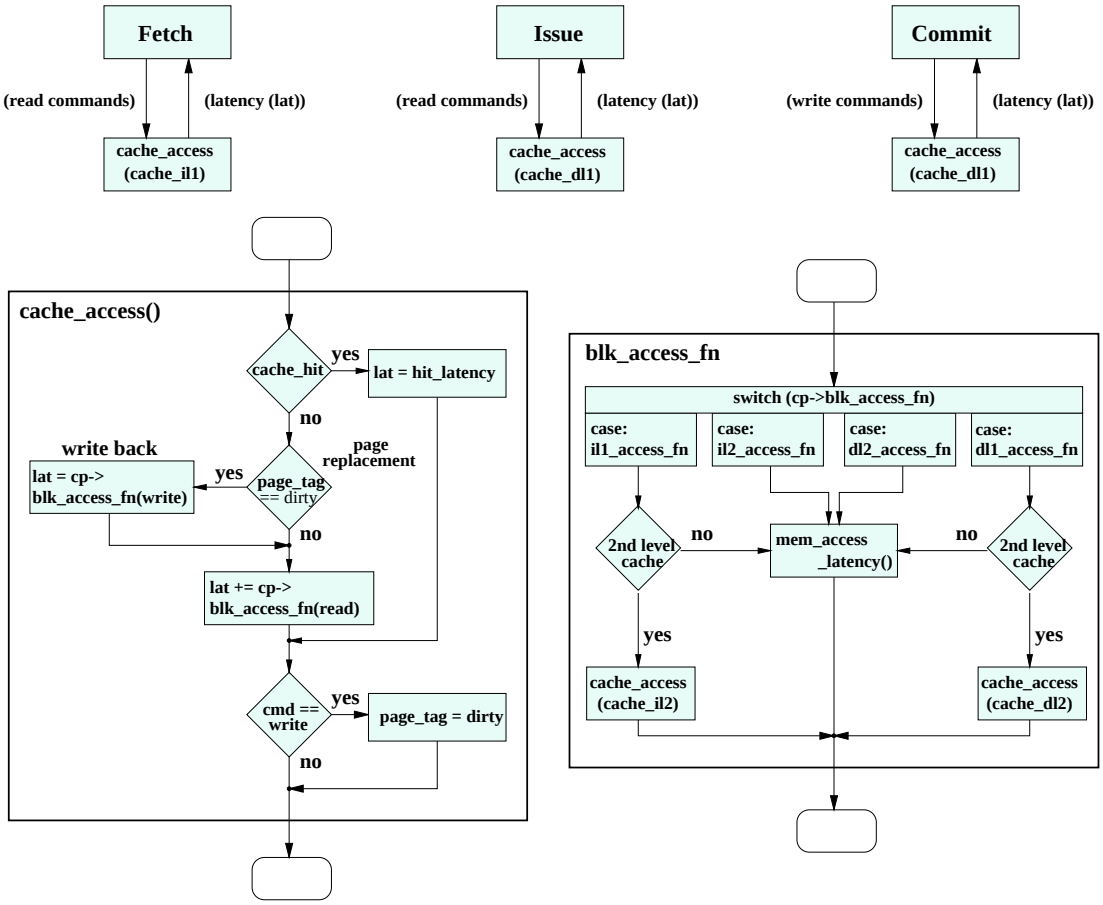

Figure 16: *Memory accesses in SimpleScalar.*

`il1_access_fn()`, `dl1_access_fn()`, `il2_access_fn()`, and `dl2_access_fn()` depending on the access type (read or write), the type of information to be accessed (program or data) and the cache level where the cache miss occurred. When there has been a second level cache miss or no second level cache exists at all, the function `mem_access_latency()` is called to compute the main memory access latency.

When a write access is initiated in the commit stage, the data will be written in the first level cache if the corresponding page is available there. The corresponding page tag is then set to dirty ,i.e., this page needs to be written back later. If a write miss occurs, the corresponding page is requested from the lower cache level and the main memory respectively. The page is fetched in the same way as in the read miss case.

Note that the memory access explained above only calculates the access latency. In the simulator, the actual data transfer is performed independently.

A.2 Implementation of the memory controller

The memory controller is implemented in the `memory_access_latency()` function. Before the memory controller is able to calculate the row and bank addresses, the virtual address must be translated to a physical address. Therefore, parts of the memory management unit are also implemented here. In order to calculate the memory region (text, heap, or

stack) the following variables and constants of the simulator are needed.

`ld_brk_point`	top address of the heap
`MD_TEXT_BAS`	base address of the text (code) segment
`MD_DATA_BAS`	base address of the heap
`MD_STACK_BASE`	base address of the stack

Whenever the virtual address is smaller than the base address of the heap, it must be an address of the text segment. If an address is larger than the address of the top of the heap, the access addresses the stack. After the calculation of the physical address the device, row, and bank addresses depending on the memory type and memory controller can be computed.

A.2.1 Address translation

SDRAM control In the SDRAM case the linear mapping memory controller computes the addresses as follows:

```
current_bank=physic_address/SD_BANK_SIZE;
current_row=(physic_address/SD_ROW_SIZE)%SD_NUM_ROWS;
```

The `physic_address` is the address calculated by the memory management unit. The interleaved mapping memory controller is implemented in the following way:

```
num_blocks = physic_address/SD_ROW_SIZE; /* row number */
current_bank=num_blocks%SD_NUM_BANKS;
current_row=num_blocks/SD_NUM_BANKS;
```

The variable `num_blocks` is the number of the addressed row. The modeled memory chip [10] contains 2^{15} rows in four banks. The constants `SD_BANK_SIZE`, `SD_ROW_SIZE`, `SD_NUM_ROWS`, and `SD_NUM_BANKS` define the size of a memory bank, the size of a memory row, the number of rows per bank, and the number of banks within a chip respectively. Since the modeled SDRAM system contains only a single DIMM, devices need not to be distinguished.

RDRAM control The linear mapping memory controller can be derived from the S-DRAM controller.

```
current_bank=physic_address/DR_BANK_SIZE;
current_row=(physic_address/DR_ROW_SIZE)%DR_NUM_ROWS;
current_device = current_bank/(DR_NUM_BANKS/DR_NUM_DEVS);
```

The interleaved mapping memory controller is implemented as follows:

```
num_blocks = physic_address/DR_ROW_SIZE; /* row number */
/* every 'second' bank will be used */
current_bank=(num_blocks%DR_NUM_BANKS)*2;
/* banks above DR_NUM_BANKS - 1 are the uneven banks */
if (current_bank >=  DR_NUM_BANKS)
   current_bank = current_bank - DR_NUM_BANKS+1;
current_row=num_blocks/DR_NUM_BANKS;
current_device = current_bank/(DR_NUM_BANKS/DR_NUM_DEVS);
```

Again, the `num_blocks` is the number of the accessed page, between 0 and $2^{16} - 1$ for the modeled RDRAM [22] system. Since the RDRAM uses shared sense amplifiers, every second bank is accessed consecutively. The banks above bank 127 are simply mapped to the odd banks (1, 3, ..., 127). This is done with the `if` statement on the fifth line. The constants `DR_BANK_SIZE`, `DR_ROW_SIZE`, `DR_NUM_ROWS`, `DR_NUM_DEVS`, and `DR_NUM_BANKS` define the size of a memory bank, the size of a memory row, the number of rows per bank, the number of devices on the channel, and the number of banks of all devices respectively.

A.2.2 Determination of access latencies

A main memory access calls the `mem_access_latency()` function with four arguments: the access command, the virtual address, the size of the access, and the simulator CPU cycle time at the start of the access. The function returns the number of CPU cycles measured from the current simulator CPU cycle time after which the memory bus will have been released.

```
static unsigned int                     /* total latency of access */
mem_access_latency(enum mem_cmd cmd,    /* access cmd, Read or Write */
                   md_addr_t baddr,     /* block address accessed */
                   int bsize,           /* block size accessed */
                   tick_t now)          /* time of access */
```

Since the function `mem_access_latency` does not consider the main memory as a shared resource, main memory accesses are non-blocking in SimpleScalar. Therefore, a semaphore called `busy_until` has been defined which states the first CPU clock cycle at which the main memory is no longer occupied by an access. Hence, only a single access may be performed at a time through the data bus. However, overlapping of control signal sequences is enabled for two consecutive accesses.

SDRAM The following timing parameters have been considered according to [10].

parameter	*description*	*clock cycles at 100 MHz*
t_{AA}	CAS latency	2
t_{RP}	precharge time	2
t_{RCD}	RAS to CAS delay	2
t_{DPL}	write data to precharge delay	2
t_{WAR}	write after read bus turn around delay	2

Closed page policy The closed page policy has been implemented without looking at a possible overlap of consecutive memory requests. Therefore, the sequence of control signals for a main memory access is always as follows: activation of the corresponding memory row, initiation of the read and write access respectively, and precharge of the row. Thus, a read always occupies the memory for eight clock cycles at memory bus clock speed using a read with automatic precharge and a burst transfer of length four. A write takes nine cycles. The transmission of the data through the memory bus however is already finished after six cycles in the write case and eight cycles in the read case respectively (see Fig. 17).

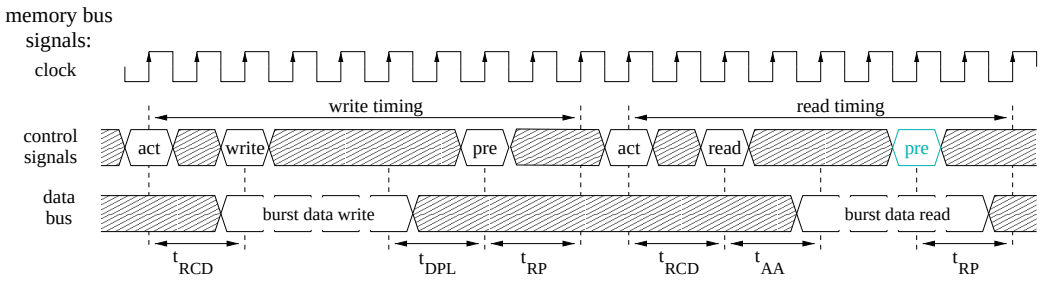

Figure 17: *SDRAM closed-page timing.*

Open page policy The open page policy can be used with or without trying to overlap parts of the control signal sequences of consecutive memory accesses. The memory controller might overlap two control sequences (we only buffer one further request) if the next memory request has been already transferred to the controller through the front-side bus while the controller is handling another request. This is possible since the modeled CPU in SimpleScalar uses a reorder buffer (the so-called *load-store queue*) for load and store instructions and therefore does not block during the execution of a load or store. Thus, the controller may initiate precharge, activation, and read accesses for the next access during the processing of the current one, see Fig. 10 for an example. However, some restrictions must be considered. Data from different accesses may not colide on the memory bus. Bus turn arounds, e.g., if a read access follows a write access, cause additional delay. Note that the controller does not change the order of accesses but only the order of some control signals to exploit the pipeline of the SDRAM. Successive accesses differ in that an access may generate a memory row hit or miss considering the preceding access. Moreover, different memory banks may be requested. Since every memory bank has its own row of sense-amplifiers, even in this situation row hits and misses may occur. Finally, the type of access, read or write, is important. Hence, 16 different cases may occur looking at two consecutive accesses plus four cases which consider the idle state of the SDRAM. However, row hits behave the same way independent on the memory bank. Thus, eight cases considering row hits have been summarized in four cases. The resulting latencies are displayed in Tab. 7. The latency is measured from the end of the current access (all burst data has been transferred through the memory bus, the corresponding memory row remains opened) to the beginning of the data transfer of the next access, i.e., for determining the end of the next transfer, the length of the burst must still be added. The *best-case* latency is achieved if the following request is transferred through the front-side bus at least (worst-case minus best-case) cycles before the end of the current access. Thus, the worst-case occurs if the next request is transferred after the end of the current one. If an open-page policy without overlapped processing is applied, the worst-case latencies must always be taken.

RAR: Read after Read, RAW: Read after Write WAR: Write after Read, WAW: Write after Write		
row hit		
type of access	*worst case access delay*	*best case access delay*
RAR	t_{AA}	0
RAW	t_{AA}	t_{AA}
WAR	$t_{WAR} - 1$	$(t_{WAR} - 1)^{a)}$
WAW	0	0

row miss in the current bank		
type of access	*worst case access delay*	*best case access delay*
RAR	$t_{RP} + t_{RCD} + t_{AA}$	$t_{RCD} + t_{AA}$
RAW	$(t_{DPL} - 1) + t_{RP} + t_{RCD} + t_{AA}$	$(t_{DPL} - 1)^{a)} + t_{RP} + t_{RCD} + t_{AA}$
WAR	$t_{RP} + t_{RCD}$	t_{RCD}
WAW	$(t_{DPL} - 1) + t_{RP} + t_{RCD}$	$(t_{DPL} - 1)^{a)} + t_{RP} + t_{RCD}$

row miss in another bank than the current		
type of access	*worst case access delay*	*best case access delay*
RAR	$t_{RP} + t_{RCD} + t_{AA}$	0
RAW	$t_{RP} + t_{RCD} + t_{AA}$	t_{AA}
WAR	$t_{RP} + t_{RCD}$	$t_{WAR} - 1$
WAW	$t_{RP} + t_{RCD}$	0

access to an idle bank / SDRAM		
type of access	*worst case access delay*	*best case access delay*
RAR	$t_{RCD} + t_{AA}$	0
RAW	$t_{RCD} + t_{AA}$	t_{AA}
WAR	t_{RCD}	$t_{WAR} - 1$
WAW	t_{RCD}	0

a) *When the previous command has already delivered its data some cycles before the next command is started, this term is zero. However, this case has not been implemented.*

Table 7: *Access latencies for the SDRAM controller given in clock cycles at 100 MHz.*

RDRAM The following timing parameters have been considered according to [22].

parameter	*description*	*clock cycles at 400 MHz*
t_{RCD}	RAS to CAS delay	7
t_{CAC}	CAS access delay	8
t_{CWD}	CAS write delay	6
t_{RP}	row precharge time	8
t_{RTR}	retire delay	8
t_{RDP}	read to precharge delay	4
t_{PP}	precharge to precharge delay	8
t_{packet}	time to transmit a packet	4

Moreover, a helper variable t_{OWR} (*overlap after write retire*) is used which is defined by $t_{OWR} = t_{packet} + t_{CWD} - t_{RTR}$.

Closed page policy The closed page policy has been implemented without looking at a possible overlap of memory instructions. Therefore, the sequence of control signals for a main memory access is always as follows: activation of the corresponding memory row, initiation of the read and write access respectively, and precharge of the row. Thus, a read always occupies the memory for 28 clock cycles and a write 31 cycles at memory bus clock speed (400 MHz) using two data packets per transfer. The transmission of the data through the memory bus however is already finished after 25 cycles in the write case and 27 cycles in the read case respectively (counted from the beginning of the activation packet). The minimal row active time t_{RAS} is kept in both cases (see Fig. 18).

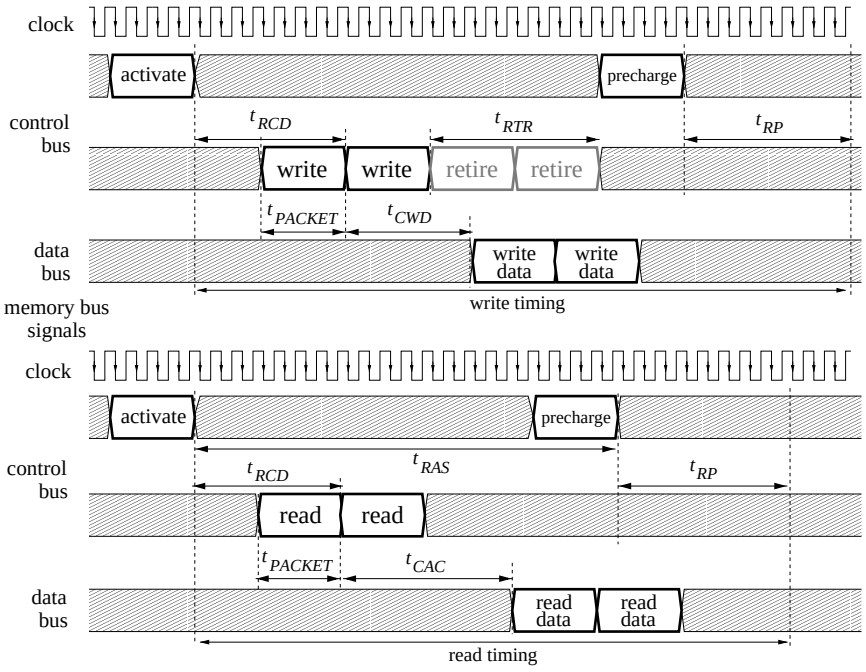

Figure 18: *RDRAM closed-page timing.*

Open page policy Successive accesses differ in that an access may generate a memory row hit or miss considering the preceding access. Moreover, different memory banks may be requested. Since sense amplifiers are shared between adjacent banks some further cases must be distinguished. For instance, assume that the next bank to be accessed is an adjacent bank to the current one. Thus, the current bank must be precharged. Therefore, changing the bank in an RDRAM may result in precharging two other banks first. Finally, the type of access, read or write, is important. Hence, 32 different cases may occur looking at two consecutive accesses to the same device plus four cases which consider accesses to different devices of the Rambus channel. The retire mechanism as well as the sharing of sense amplifiers are responsible for the higher case count than in the SDRAM variant. The resulting latencies are displayed in Tab. 8. The latency is measured from the end of the current access (all burst data has been transferred through the memory bus, the corresponding memory row remains opened) to the beginning of the data transfer of the next access, i.e., for determining the end of the next transfer, the length of two packets must still be added. The *best-case* latency is achieved if the following request is transferred

through the front-side bus at least (worst-case minus best-case) cycles before the end of the current access. Thus, the worst-case occurs if the next request is transferred after the end of the current one. If a open-page policy without overlapped processing is applied, the worst-case latencies must always be taken.

RAR: Read after Read, RAW: Read after Write, WAR: Write after Read, WAW: Write after Write
sd: same device, od: other device

row hit

type of access	*worst case access delay*	*best case access delay*
RAR	$t_{PACKET} + t_{CAC}$	0
RAW sd	$t_{PACKET} + t_{CAC}$	$t_{PACKET} + t_{CAC} - t_{OWR}$
RAW od	$t_{PACKET} + t_{CAC}$	$t_{CAC} - t_{CWD}$
WAR	$t_{PACKET} + t_{CWD}$	0
WAW	$t_{PACKET} + t_{CWD}$	0

row miss in the current bank

type of access	*worst case access delay*	*best case access delay*
RAR	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{RCD} + t_{CAC}$
RAW	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC} - t_{OWR}$
WAR	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CWD}$	$t_{RCD} + t_{CWD}$
WAW	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CWD}$	$(t_{RP} - t_{OWR}) + t_{RCD} + t_{PACKET} + t_{CWD}$

row miss in another bank than the current, bank already active

type of access	*worst case access delay*	*best case access delay*
RAR	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	0
RAW sd	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{PACKET} + t_{CAC} - t_{OWR}$
RAW od	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{CAC} - t_{CWD}$
WAR	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CWD}$	0
WAW	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CWD}$	0

access to an idle bank, adjacent banks idle

type of access	*worst case access delay*	*best case access delay*
RAR	$t_{RCD} + t_{PACKET} + t_{CAC}$	0
RAW sd	$t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{PACKET} + t_{CAC} - t_{OWR}$
RAW od	$t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{CAC} - t_{CWD}$
WAR	$t_{RCD} + t_{PACKET} + t_{CWD}$	0
WAW	$t_{RCD} + t_{PACKET} + t_{CWD}$	0

Access to an idle bank, adjacent banks active
p: necessary precharges for access, a: bank to access is adjacent bank of the current bank, !a: not a

type of access		*worst case access delay*	*best case access delay*
R	p=1, !a	equivalent to RAR, row miss in another active bank	
A	p=1, a	equivalent to RAR, row miss in the same bank	
R	p=2, !a	$t_{PP} + t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	0
	p=2, a	$t_{PP} + t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{RDP} + t_{RP} + t_{RCD} - t_{PACKET}$
R	p=1, !a	equivalent to RAW, row miss in another active bank	
A	p=1, a	equivalent to RAR, row miss in the same bank	
W	p=2, !a, sd	$t_{PP} + t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{PACKET} + t_{CAC} - t_{OWR}$
	p=2, !a, od	$t_{PP} + t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{CAC} - t_{CWD}$
	p=2, a	$t_{PP} + t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC}$	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CAC} - t_{OWR}$
W	p=1, !a	equivalent to WAR, row miss in another active bank	
A	p=1, a	equivalent to RAR, row miss in the same bank	
R	p=2, !a	$t_{PP} + t_{RP} + t_{RCD} + t_{PACKET} + t_{CWD}$	0
	p=2, a	$t_{PP} + t_{RP} + t_{RCD} + t_{PACKET} + t_{CWD}$	$t_{RDP} + t_{RP} + t_{RCD} + t_{CWD} - t_{PACKET} - t_{CAC}$
W	p=1, !a	equivalent to WAW, row miss in another active bank	
A	p=1, a	equivalent to RAR, row miss in the same bank	
W	p=2, !a	$t_{PP} + t_{RP} + t_{RCD} + t_{PACKET} + t_{CWD}$	0
	p=2, a	$t_{PP} + t_{RP} + t_{RCD} + t_{PACKET} + t_{CWD}$	$t_{RP} + t_{RCD} + t_{PACKET} + t_{CWD} - t_{OWR}$

Table 8: *Access latencies in clock cycles at 400 MHz for the RDRAM controller.*