A MIXED QOS SDRAM CONTROLLER FOR FPGA-BASED HIGH-END IMAGE PROCESSING

Dipl.-Ing Sven Heithecker, Dipl. Ing. Amilcar do Carmo Lucas, Prof. Dr.-Ing. Rolf Ernst,

Technical University of Braunschweig
Institute for Computer and Communication Network Engineering
heithecker,lucas,ernst@ida.ing.tu-bs.de

ABSTRACT

High-end video and multimedia processing applications today require huge amounts of memory. For cost reasons, the usage of conventional dynamic RAM (SDRAM) is preferred. However, accessing SDRAM is a complex task, especially if multi-stream access, different stream types and realtime capability are an issue. This paper describes a multi-stream SDRAM controller IP that covers different stream types and applies memory scheduling to achieve high bandwidth utilization. Two different architectures are presented and discussed, simulation results with a realistic application configuration demonstrate up to 90% of maximum memory bandwidth utilization. The scheduler IP is suitable for FPGA implementation and is flexible enough to be used in other applications.

1. INTRODUCTION

Dynamic RAM memories are important components in multimedia and embedded systems. For example, they are used to store large frames in multimedia and video applications. In high-end applications, such as HDTV or electronic motion pictures, bandwidth is critical. High resolution applications, widely used in motion picture and advertising industries require up to 2K[1] resolutions that translate to a data-rate of 2.1 Gbit per second and channel [2], [1].

The very high end of digital cinema (D-Cinema) applications have grown in importance over the last couple of years with a brilliant resolution of 4K per frame [3] and even higher resolutions are to be expected in the future. Real-time processing, such as filtering for up-/and down-scaling, color keying, compression or trick effects at this data rate and precision is beyond the scope of today's workstations and single DSP processors. The market volume for such systems is very small, so ASICs are not economically viable and therefore not an option. Therefore, we follow an FPGA-based system approach as shown in figure 1.

The main problem of DRAM architectures is the long access latency. Current DRAM architectures (DDRAM, DirectRambus, ...) reduce the overhead due to this latency using burst access to several consecutive data words in a memory row. The effect of access latency can be further reduced by exploiting the internal bank structure of a DRAM and switching to another bank while one bank is busy accessing. This technique, called interleaving, increases memory bandwidth, however without reducing the individual memory access latency. If there are enough banks, then the burst length and, hence, the individual access latency can be reduced without significant loss in memory bandwidth. This patented[**?**] technique has already been utilized in the FPGA based memory controller of an commercial HDTV mixer application reaching on average 70% of the SDRAM peak performance with a burst length of only 4 words [7].

With even higher throughputs and more complex access patterns, the latency grows again due to larger intermediate buffers and a higher scheduler complexity. In this paper, we propose a two stage controller architecture consisting of a bank scheduler that is adapted to the memory structure and a request scheduler that reflects the access patterns. We investigate two different buffering and scheduling strategies that are simple enough to run in realtime.

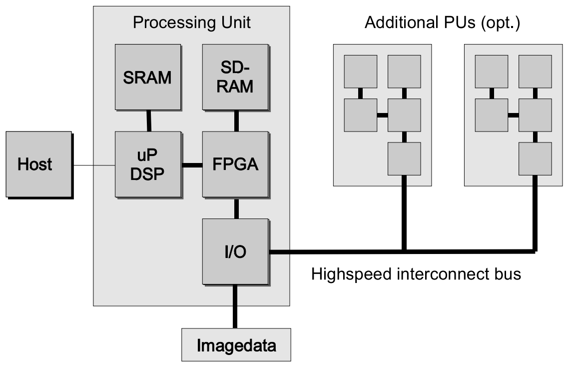

Fig. 1. Flexible image processing platform

After a concise introduction to requirements and related work the two variants of the 2-stage memory controller architecture are described in section 3. Section 4 describes the SystemC based evaluation environment, the experimental setup and the results. Section 5 draws a conclusion.

2. REQUIREMENTS AND RELATED WORK

A closer look reveals that the following distinct types of accesses have to be served by the SDRAM controller:

- Hard vs. soft realtime. While for some access sequences a maximum lantency must be guaranteed, e.g. image streams which must not lose any pixel (hard real-time), other access sequences like user-interface CPU accesses might not suffer

[1]This resolution means 2048x1556 pixels per frame at 30 bit/pixel and 24 pictures/s resulting in 11.4 Mbytes memory requirements per frame

from a short stall (soft real-time). However, it has to be ensured that no access sequence gets completely blocked.

- Fixed address patterns, like mage input/output streams or data streams generated by FPGA-based processing accelerators. To increase troughput and to compensate the burst-oriented SDRAM access, data prefetching and large FIFOs which are needed for higher troughput can be used. Since these are usually hard real-time streams, a *maximum latency* and *minimum troughput* must be guaranteed.
- Random address patterns, like DSP memory access in case of cache miss. Since these address patterns are nondeterministic, data prefetching cannot be used. Instead, to minimize stall times, these accesses should be served with *smallest possible latency*, therefore small buffers and short burst lengths should be used. Depending on the CPU task, these access sequences can be hard or soft real-time.

This results in two types of QOS: *guaranteed minimum troughput at guaranteed maximum latency* and *smallest possible latency*.

The Imagine processor [11] uses a configurable memory scheduler [4][10], optimized for the application algorithms that run on the processor. The scheduler is adapted to a specific application and does not distinguish different stream types (hard-real time vs. soft-real-time). The Prophid architecture [6] by Meerbergen et al. describes a dynamic RAM scheduler for the Prophid DSP platform that is focused on streams using large FIFO buffers and round-robin scheduling. The "Center for Embedded Computer Systems" group at UCI [9] provides optimization heuristics for known memory access patterns of a single processor. None of these schedulers support vastly different access types at close to peak SDRAM bandwidth.

Finally, Sonics offers a memory scheduler IP [13]. Albeit they support different QOS types, their implementation is very different to our solution. While our solution consists of an 2 stage scheduler with internal buffering, Sonics uses a 7 stage filter without buffering. Due to the higher complexity, the Sonics IP is targeted at ASICs at $\sim$ 100 MHz, while our solution is suitable for FPGAs at $\sim$ 140 MHz.

3. ARCHITECTURE

Our main design goal was to reach maximum memory bandwidth for multiple access streams with different access sequence types running in parallel. As explained in the introduction, the two objectives (maximum bandwidth and minimum latency) require contradictory buffer types and memory burst lengths.

Fig.2 shows the overall memory controller structure with the two scheduler variants, one with different acess pathes and distinct FIFOs for different priorities, and one, slightly less complicated architecture with merged access pathes and FIFOs. In this paper, we focus on the two blocks, request scheduler and bank scheduler, their respective buffers and their scheduling strategies.

3.1. 1st Variant: Request Priority Scheduling

3.1.1. Request Scheduler, Bank Buffer

Requests to the SDRAM controller are always made at full burst length. The request scheduler forwards requests of several inputs with different request types, one request per clock cycle, to the bank buffer. The bank buffer FIFOs are small FIFOs, one per bank, which store the requests sorted by bank.

By applying a priority-based round-robin arbitration similar to [12], it is guaranteed that successively each input gets scheduled (assuming a request is available and the according bank buffer is empty). Requests from high priority inputs are scheduled out of order before requests from standard inputs. To prevent standard priority requests from being blocked by continuous high priority requests, after a consecutive scheduling of all high priority inputs one standard priority request gets scheduled. Without loss of generality, we use 2 priority levels.

3.1.2. Bank Scheduler

The bank scheduler is responsible for selecting requests from the bank buffer and forwarding them to the tightly coupled access controller for execution. In order to increase bandwidth utilization, two aspects have to be considered: *bank interleaving* and *request bundling*.

Bank interleaving is used to hide the bank access latencies. In a common SDRAM setup, an access to a bank in auto-precharge mode and with burstslength 4 takes 4 active cylces in which data is transferred, follwed by 4 (read) to 6 (write) passive cycles during which the bank cannot be accessed. However, during these passive cycles, another bank can be accessed. Thus, with at least 3 banks available, it can be guaranteed that each memory bank can be accessed once in a round without stalling. The scheduler maintains bank busy flags which describe the current state of the bank. By only selecting banks which are not busy and by applying a priority-based round-robin arbitration similar to [12], it is guaranteed that successively each bank gets scheduled (assuming a request is available).

Request bundling is used to minimize bus direction switches. On every bus direction switch, tristate cycles have to be inserted (1 for a read-write change, 1-2 for a write-read change, depending on the type of SDRAM) which can lead to a bandwidth decrease of up to 20..27% for alternating read-write accesses. Bundling requests to consecutive read or write sequence alleviates this. To prevent deadlocks, only one request of each bank is allowed in one read or write bundle.

Note that the handling of the bank busy states is a major difference to the Sonics solution. While they filter out busy banks at the beginning of their scheduling strategy, we first schedule requests by priority, store them in FIFOs, and later select them from these FIFOs depending inter alia on the bank busy state.

3.2. 2nd Variant: Bank Priority Scheduling

3.2.1. Request Scheduler, Bank Buffer

In this variant, independent pathes are provided for different request priorities. Each priority has got its own request scheduler and their own bank buffer FIFOs. Since the same arbitation scheme (without request priorities) is used as in the 1st variant, we still have a deterministic maximum latency and per-input troughput. As expained above, different bank buffer FIFOs for different priorities exist. A later multiplexor selects high priority requests if available, otherwise standard priority requests. To prevent normal requests from being deadlocked by continuous high priority requests, after a sequence of high priority requests one normal request will be selected if available. This is controlled by the Bank Scheduler.

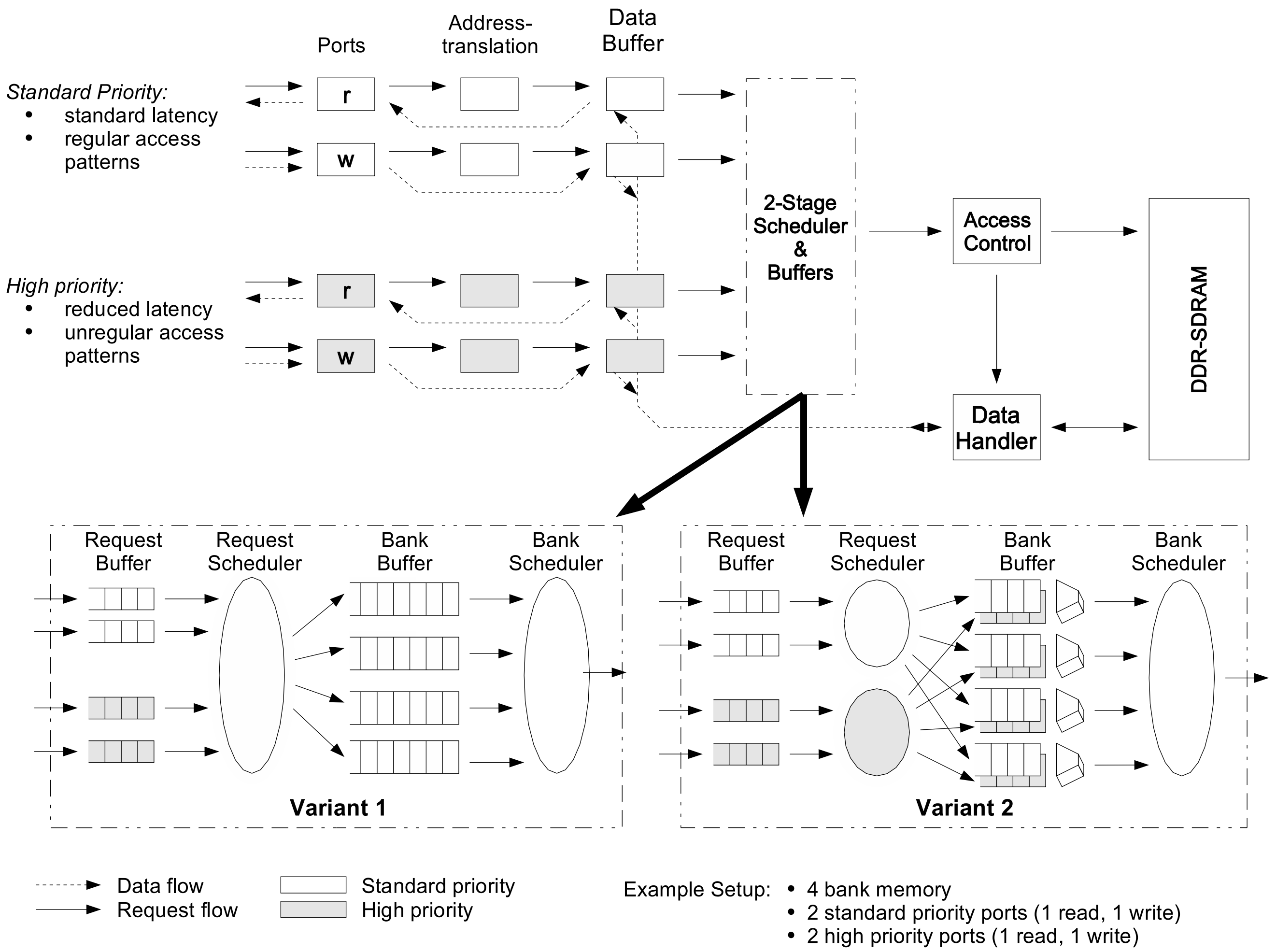

Fig. 2. SDRAM controller architecture

The additional complexity for the scheduler is moderate since the complexity of each scheduler is less compared to the 1st variant (less inputs and no priority selection logic). The bank buffer complexity increases noticeable, especially if more than two priorities are used, however it might be possible to reduce each FIFO size.

3.2.2. Bank Scheduler

The bank scheduler works similar to the 1st variant regarding bank interleaving, request bundling and the round-robin arbitration scheme. In addition, the priority handling has moved from the request scheduler to the bank scheduler. To accomplish this, high priority requests are given precedence over standard priority requests. Only if no high priority requests can be scheduled, a standard priority request is taken. To prevent standard priority requests from being blocked by continuous high priority requests, after a consecutive scheduling n_{banks} high priority requests one standard priority request gets scheduled if available (in this case, the bank buffer FIFO multiplexors select a standard priority request regardless of available high priority requests as stated above).

3.3. Additional Modules

Since the rest of the SDRAM controller is the same in both variants, they will be explained only once.

3.3.1. Access Controller

The Access Controller is responsible for SDRAM command issuing, including initialization and refresh cycles. As stated above, and contrary to most other SDRAM controllers, the SDRAM is accessed using auto-precharge mode, which means that a bank is automatically precharged after an access. This results in deterministic bank throughput and access latency which is needed for a global real-time analysis. Also, the complexity of the controller decreases due to a simpler state machine and lack of row address comparators, which makes it more suitable for FPGAs. The Access Controller works with a burst length of 4 words (8 words on DDR-SDRAM).

3.3.2. Address Translation

Due to the auto precharge mode, there is no advantage in mapping access sequences to the same bank and row to exploit row buffer hits as with precharge-based SDRAM controllers [14]. In contrast, the latency increases if two consecutive requests hit the same bank. To avoid this, request sequences are spread over all banks by permutating and *xor*-ing original address bits analogous to [15]. For many deterministic address patterns, e.g. linear address incrementation during video input, the optimum bank access sequence can be achieved by adapting the algorithm.

3.3.3. Data Buffer

The write-request data gets stored inside data buffers. Once the request has been scheduled, the data is transferred to the SDRAM. For read requests, an empty place in the according data buffer is assigned to the request. If the read is executed, the data from the SDRAM is transferred to that place. The bank buffer access is done via tags which are assigned to the requests. This technique allows an easy adaption of the off-chip SDRAM databus width and the usually smaller internal datapathes. It also alleviates the need to store the data inside the buffer FIFOs and the buffer-to-buffer transfers.

3.4. Real-Time Aspects

Since all schedulers use round-robin arbitration, and deadlocks due to continuous high-priority requests are prevented by temporarily forcing the scheduling of low-priority requests, we have a deteministic maximum latency and minimum troughput.

3.5. Parameters

The controller is parameterizable to serve different applications. The most important parameters are the SDRAM layout (rows, columns, banks) and timing, number of inputs including their priority and the FIFO sizes. Those parameters directly influence the latency and throughput. Increasing the number of inputs increases the maximum per-port latency and decreases the per-port maximum troughput (decrease of performance). Increasing the number of banks also increases the per-bank latency and decreases per bank troughput which in turn also affects the worst case performance. However, more banks also increases the possibility that two requests get mapped to different banks which leads to a better average performance. However, if the bank access sequence can be predicted as stated in 3.3.2, then also the worst-case performance is improved.

4. EXPERIMENTS

To evaluate the concept, we created a system model consisting of the SDRAM, SDRAM controller as well as several clients. We used $SystemC$ for implementation because, besides the high simulation speed, it allowed us to model at serveral abstraction levels in one language and to easily reuse available C models.

For speed reasons, the SDRAM controller and SDRAM model were implemented at cycle-accurate transaction level, except for the interface between controller and SDRAM which is at RTL. The complex SDRAM timing was modeled precisely to yield meaningful results. For the SDRAM we used the same parameter set (table 1) throughout all experiments.

parameter	value
technology	SDR
clock	142 MHz
size	64 MByte
banks	8
databus width	32 bits
burstlenth	4
$t_{rcd}, t_{cl}, t_{rp}, t_{wr}$	2
t_{rc}	8
refresh	8192 auto-refresh per 64ms

Table 1. SDRAM parameter

4.1. Experiments 1a,1b

To test the impact of the bank buffer size regarding maximum troughput and latency, we created one random access, 50% read/write data stream and did serveral runs with different loads and bank buffer sizes. We used architecture variant 1, however there is no difference to variant 2 because there are no priorities involved (experiments not shown). We recorded the yieled troughput and latency. Table 2 shows the setup and the results. We made the following observations:

- the maximum reachable SDRAM troughput is 0.9 words/cycle, which is a bandwith utilization of 90%.
- increasing the buffer sizes has no major effect on the troughput, but it increases the latency significantly on high loads due to FIFO fill up. On the other side, with 8 banks available, there are enough banks left to achieve maximum troughput even if one bank buffer is empty due its small size. Therefore, a buffer size of 1 is ok.

With the small bankbuffer FIFO size of 1, it is interesting to see if architecture variant 2 with its distinct pathes for high and low priority requests provides better QOS service than architecture variant 1. To verify this, we created a 2nd access stream which we gave a higher priority. We tested several load combinations (wich together do not exceed the maximum reachable troughput) with both architecture variants and recorded the yielded troughput and the latency (table 3).

- We clearly see that effect of the priorities in architecture variant 2 is bigger than in architecture variant 1, that means that the latency of the high priority stream is lower while the latency of the low priority stream is higher. That means that even with small buffers, it is better to have different request pathes and merge the requests as late as possible. Since the bank buffer FIFO size is only 1, the added complexity is moderate and thus acceptable.
- The priority architecture has no impact on the maximum troughput. Both streams reach their requested troughput.

4.2. Experiment 2

While the synthetic experiment 1 gives a good impression of the memory controller performance, we are also interested in the performance under practical conditions. For this purpose, we defined an example architecture which ressembles the setup in [7] and implemented an example application. The experimental setup is shown in figure 3. It consists of a hardware implementation of a discrete wavelet transformation (DWT) algorithm and a CPU with caches.

applied	BB size 1		BB size 2		BB size 5		BB size 10	
load	tp	lat	tp	lat	tp	lat	tp	lat
[w/c]	[w/c]	[cycles]	[w/c]	[cycles]	[w/c]	[cycles]	[w/c]	[cycles]
0.8	0.80	11.3	0.80	11.5	0.80	11.5	0.80	11.5
0.9	0.90	29.9	0.90	32.7	0.90	32.9	0.90	31.0
0.95	0.92	39.0	0.92	54.1	0.92	87.7	0.93	139.6
1.0	0.90	41.7	0.92	54.1	0.92	101.1	0.91	176.3

BB size - bank buffer FIFO size
w/c - words per cycle
tp - troughput
lat - latency

Table 2. Experiment 1a: Simulation setup and results

applied	Architecture variant 1		Architecture variant 2	
load	troughput	latency	troughput	latency
lo / hi	lo / hi	lo / hi	lo / hi	lo / hi
[w/c]	[w/c]	[cycles]	[w/c]	[cycles]
0.2 / 0.5	0.2 / 0.5	11 / 9	0.2 / 0.5	9.5 / 9.5
0.2 / 0.7	0.2 / 0.5	45 / 21.5	0.2 / 0.5	30 / 30.5
0.5 / 0.2	0.5 / 0.2	10 / 9	0.5 / 0.2	9.5 / 9.5
0.7 / 0.2	0.7 / 0.2	35.5 / 15.5	0.7 / 0.2	29.5 / 22.5

w/c - words per cycle
lo - low priority stream
hi - high priority stream

Table 3. Experiment 1b: Simulation setup and results

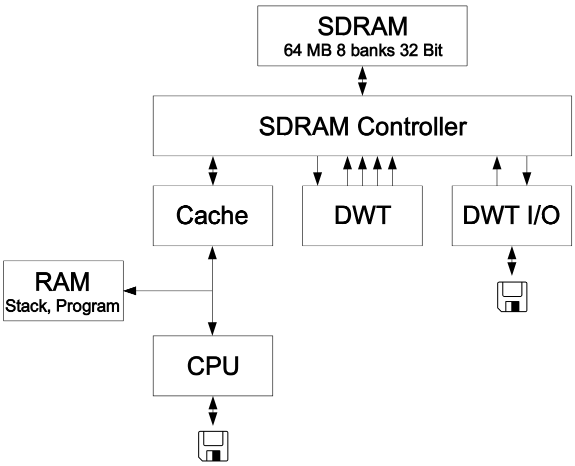

Fig. 3. Experimental setup

4.2.1. Discrete Wavelet Transformation

The DWT implementation is based on [8] and was implemented in $SystemC$ at the RT-Level to obtain a cycle true simulation model and a basis for a precise cost and performance estimation via logic synthesis.

After synthesis followed by place and route for a Xilinx VirtexII FPGA, the CAD tools reported a maximum operating frequency of 100 MHz. This frequency is not a integer fraction of the expected 140 MHz SDRAM memory interface clock. So in order to achieve maximum throughput we used asynchronous FIFOs to transfer data between the two different clock domains. The DWT input consumes one data item (2 pixels) per clock cycle, while the four outputs are idle for 8 cycles, followed by 8 cycles of data output through 4 parallel write ports.

The period of the output streams depends on the image width and on the current DWT level [8]. Because this algorithm, as well as most other image transform or filtering algorithms, have a known memory access pattern, we are able to apply pre-fetching to all input data. This is aided by the fact that the avaliable FIFOs on the FPGA are relatively large. This way, the fixed access pattern can be served by burst memory access.

The data I/O for the DWT uses a separate port, DWT I/O. This port reads and writes a periodic pixel stream.

4.2.2. CPU

For CPU simulation, the SimpleScalar SimSafe DLX simulator together with an adaptation of the Dinero cache simulator were used. The CPU has its own fast local RAM for program code and local variables (stack) and uses the SDRAM for larger data storage.

In the final system, the CPU will be replaced by a DSP, but this architecture is sufficient to generate meaningful burst cache access patterns for these experiments.

4.2.3. Simulation Results

The DWT was setup for a three level 512 x 512 grayscale image, and the CPU performed a compression of a 64 x 64 color image using cjpeg from the media testbench [5]. The SDRAM controller was clocked at 140 MHz, the DWT at 100 MHz, which the CAD tools indicated is reachable with our FPGA configuration. The CPU and the cache (2Kb, 4-way associative, block size equal to SDRAM burst length) were clocked at 1 GHz to compensate for the weak DLX performance (as said in Subsection 4.2.2 ideally we would use a powerful DSP instead). We measured the execution time of the DWT and the CPU for several runs with the following parameters:

- Running only CPU, only DWT and both CPU and DWT.
- Bank buffer FIFO size of 1 vs. 6. For architecture 2, we set a) both high and standard priority FIFOs to 6 (2x6) and b) both high and standard priority FIFOs to 3 (2x3). The later (2x3) sizes have about the same complexity as a single FIFO of size 6.

Nr.	DWT	CPU	BB size	Architecture 1 DWT Kcycles	Architecture 1 CPU Mcycles / CPI	Architecture 2 DWT Kcycles	Architecture 2 CPU Mcycles / CPI
1	no	yes	1		5.75 (1.89)		5.75 (1.89)
2	no	yes	6 / 2x6		5.75 (1.89)		5.75 (1.89)
3	no	yes	6 / 2x3				5.75 (1.89)
4	yes	no	1	504		504	
5	yes	no	6 / 2x6	504		504	
6	yes	no	6 / 2x3			504	
7	yes	yes	1	531	6.75 (2.22)	531	6.61 (2.17)
8	yes	yes	6 / 2x6	532	6.76 (2.22)	538	6.63 (2.18)
9	yes	yes	6 / 2x3			538	6.63 (2.18)

BB size - bank buffer FIFO size

Table 4. Experiment 2: Simulation setup and results

The results are shown in table 4. We made the following observations:

- Architecture 2: setting the bank buffer FIFO size to 2x3 shows no difference compared to the 2x6 setting (nr. 2,5,8 vs. 3,6,9)
- The mutual influence of CPU activity and DWT is small, despite the relatively large load and despite the very different access patterns. This is a result of the flexible 2 stage scheduling scheme with short access bursts and small round-robin time slots leading to low latencies.

5. CONCLUSION

Based on earlier experience with a fixed architecture for a reconfigurable HDTV system, we presented two variants of a dynamic RAM scheduler IP that supports several concurrent access sequence types with different requirements including hard real-time periodic sequences and cache accesses with a minimum latency objective. It consists of a 2-stage scheduler, a Bank Scheduler for memory efficiency optimization and a Request Scheduler to arbitrate the access streams. We explained the two variants, the chosen IP parameters and their impact on design performance. The IP has been evaluated in a simulation environment consisting of a processor with cache, an application specific data path for wavelet coding and video I/O, all implemented in SystemC. In the evaluation, we demonstrated the high flexibility and efficiency of the 2-stage approach. The simulation data show a 90% memory bandwidth utilization and adherence to access requirements for a wide range of load scenarios. The IP can easily be adapted to DDR-RAM or RAMBUS DRAM by simply exchanging the memory interface and the bank model.

6. REFERENCES

[1] http://www.discreet.com.

[2] http://www.quantel.com.

[3] http://www.thomsonbroadcast.com.

[4] Brucek Khailany, William J. Dally, S. R. Imagine: Media processing with streams. *IEEE Micro* (March/April 2001), 35–46.

[5] Chunho Lee, Miodrag Potkonjak, W. H. M.-S. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture* (1997), pp. 330–335.

[6] Jeroen A. J. Leitjen, Jef L. van Meerbergen, A. H. T. Prophid: A heterogeneous multi-processor architecture for multimedia. In *International Conference on Computer Design* (October 1997), pp. 164–169.

[7] Kersten Henriss, Peter Rueffer, R. E. A reconfigurable hardware platform for digital real-time signal processing in television studios. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines* (April 2000).

[8] Po-Cheng Wu, L.-G. C. An efficient architecture for two-dimensional discrete wavelet transform. *IEEE Transactions on circuits and systems for video technology 11*, 4 (April 2001).

[9] Prabhat Mishra, Peter Grun, N. D. Processor-memory co-exploration driven by a memory-aware architecture description language. In *14th International Conference on VLSI Design* (Jan 2001).

[10] Scott Rixner, William J. Dally, U. J. K. Memory access scheduling. In *International Symposium on Computer Architecture* (2000), pp. 128–138.

[11] Ujval J. Kapasi, William J. Dally, S. R. The imagine stream processor. In *International Conference on Computer Design* (2002).

[12] Weber, M. Arbiters: Design ideas and coding styles. In *SNUG, Boston* (2001).

[13] Weber, W.-D. Sonics memmax memory scheduler.

[14] Wei-Fen Lin, Steven K. Reinhardt, D. B. Designing a modern memory hierarchy with hardware prefetching. *IEEE Transactions on Computers 50*, 11 (November 2001), 1202–1218.

[15] Zhao Zhang, Zhichun Zhu, X. Z. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *International Symposium on Microarchitecture* (2000), pp. 32–41.