A Stream-Type Aware SDRAM Controller for FPGA-Based High-End Video Editing

XXXXXXXXXXXXXXXXXXXX

XXX XXX XXX
XXX XXX XXX
XXX XXX XXX

ABSTRACT

This paper describes a dynamic memory scheduler IP that is suitable for FPGA implementation. It is targeted at high-end multimedia applications with different access patterns. Results with a realistic application configuration demonstrate 90% of maximum memory bandwidth utilization. The scheduler IP is flexible enough to be used in other applications.

1. INTRODUCTION

Dynamic RAM memories are important components in multimedia and embedded systems. They are used to store routing tables in network processors or large frames in multimedia and video applications. In high-end applications, such as HDTV or electronic motion pictures, memory cost and bandwidth are 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. Two approaches remain:

- using parallel DSP processors with interleaved memory access,
- using large FPGAs for highly regular algorithms extended by DSP processors for less regular applications.

The first option requires many powerful DSPs to achieve the desired results, which leads to complex and expensive systems. The second option explores the fact that today FPGAs have a very large amount of logic that can implement multiple arithmetic operations per clock-cycle. FPGAs can be used to implement the more regular (less control intensive) parts of the image processing algorithms, leaving the more control intensive parts to be implemented on DSPs. We followed this second philosophy

[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

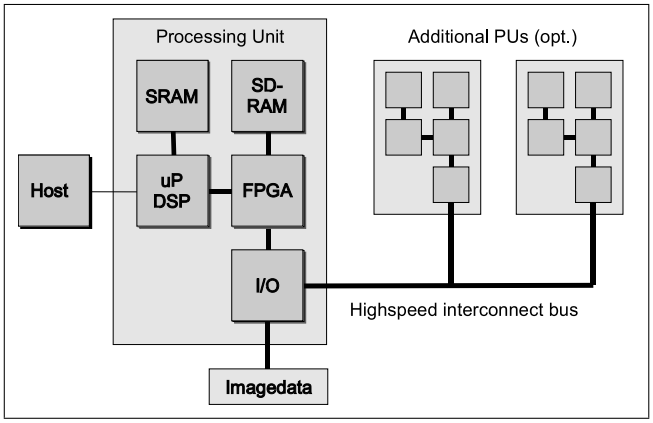

Figure 1: Flexible image processing platform

in a previous work when we developed the FPGA-centric architecture for a commercial HDTV mixer application. The architecture used is shown in Figure 1. To solve the memory access bottleneck of this kind of application, a specialized low-latency memory scheduler was developed [6] which uses short burst accesses that enable a 70% SDRAM bandwidth utilization. It was implemented on a $VirtexI$ ® FPGA.

However, to achieve higher resolutions and to be able to implement even more complex algorithms, we propose to extend our previous approach by making the memory controller stream type aware. A closer look reveals that three distinct types of accesses have to be served:

- hard real-time data streams with constant throughput. The input and output image streams are strictly periodic and must not lose any pixel data.
- data streams with hard real-time throughput and fixed periodic access patterns. These streams are found at the interface to the hardware accelerators attached to the FPGA.
- a best effort access with a minimum latency cost function. The DSP accesses the memory in case of cache misses or DMA access. This access should be served with minimum latency to minimize stall times. Best effort is typically sufficient.

In this paper, we present a parameterized dynamic RAM scheduling IP that covers multiple stream types and scheduling strate-

gies. After a concise introduction to related work, the memory scheduler IP is described in section 3. The IP, and all other parts, are modeled using *SystemC* with a transaction based modeling style. This is explained in section 4. Using a realistic application mix, we demonstrate improved performance of more than 90% maximum bandwidth in section 5. The paper concludes with a summary and an outlook on future work.

2. RELATED WORK

The Imagine processor [11] uses a configurable memory scheduler [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 [7] 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 group of UCI [5] provides optimization heuristics for known memory access patterns of a single processor. Finally, Sonics offers a memory scheduler IP [4] for their specific TDMA bus protocol.

3. ARCHITECTURE

Our main design goal was to reach maximum memory bandwidth for different access sequence types running in parallel. As explained in the introduction, the maximum bandwidth objective is a contradictory objective to the minimum latency requirement for the DSP. On the other hand, real-time stream accesses can be guaranteed by bounding the maximum latency time.

The only way to obtain low latency times is to minimize the memory access burst length. Short bursts, however, increase memory access overhead. This can be reduced by using several memory banks interleaving bank access. Therefore, the two concerns, memory bandwidth optimization at low latency and access scheduling are separated and controlled by two distinct schedulers, a Bank Scheduler and a Request Scheduler. The Bank Scheduler has a known maximum latency time, and the Request Scheduler can serve both low latency and real-time requests, such that the chain of both schedulers can serve access sequences with low latency objectives and, at the same time, access sequences with hard real-time requirements.

Our proposed architecture is shown in Figure 2. The controller uses the auto-precharge mode for accessing the SDRAM, which means that a bank gets automatically precharged after an access. This results in deterministic bank throughput and access latency which is needed for a global performance analysis.

3.1 Bank Scheduler

The first scheduler, the Bank Scheduler, is responsible for bank scheduling to achieve high throughput and, together with the access controller, for SDRAM access command issuing. In order to increase bandwidth utilization, two aspects have to be considered: bank interleaving and request bundling.

Bank interleaving is used to increase bandwidth utilization. Table 1 shows that for $bl = 4$, one access takes 4 clock cycles, followed by 4 to 6 (read, $t_{rp}+t_{rcd}$) or 6 to 9 (write, $t_{wr}+t_{rp}+t_{rcd}$) clock cycles of inactivity, respectively. Thus, with bank interleaving

$$n_{banksMin} = \left\lceil \frac{bl + t_{wr} + t_{rp} + t_{rcd}}{bl} \right\rceil = 3 \ldots 4 \text{ for } bl = 4$$

Parameter	Description	common values in clock cycles
bl	burst-length in words	1,2,**4**,8
t_{rcd}	RAS-to-CAS-delay	2,3
t_{cl}	column latency	2,3
t_{wr}	write recovery	2,3 (usually = t_{cl})
t_{rp}	row precharge	2,3
t_{cbr}	cycles between refresh	2078
t_{rc}	row cycle time	12

Table 1: Important SDRAM parameters

banks are needed for maximum throughput. The scheduler maintains the internal state of every bank and assigns scheduler priorities to all banks. The bank with the highest scheduler priority which is not busy is selected for the next access. After every access, the scheduler priorities are rotated so that the bank accessed last gets the lowest priority, as proposed in [12] . This mechanism allows interleaved bank accesses as well as a guaranteed access latency, because every bank will get the highest priority at least once in a cycle of n_{bank} accesses, where n_{bank} is the number of banks. 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 [13]. In contrast, if two consecutive requests of one access sequence hit the same bank, the latency increases due to the precharge-activate cycle that the bank has to undergo first before a new access can be issued. However, if those two requests go to different banks, they can be scheduled back-to-back without additional latency. Therefore, request sequences have to be spread over all banks. This is done by permutating and *xor*-ing original address bits analogous to [14]. This is done in the initial address translation stage just after the SDRAM controller inputs.

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, $t_{cl} - 1$ for a write-read change) 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.

Low latency requests are given precedence over normal requests so that they are scheduled first, if possible. However, to prevent normal request from being deadlocked by continuous low latency requests, only one low latency request per bank is executed during one sequence of continuous low latency requests. Between successive low latency sequences, one normal request can be executed if available.

3.2 Request Scheduler

The second scheduler is the Request Scheduler who forwards requests of several input streams, one request per clock cycle, to their appropriate bank buffer FIFOs. It works similar to the Bank Scheduler by assigning scheduler priorities to the inputs and trying to schedule inputs with a higher scheduler priority first. There is a priority rotation scheduling scheme that guarantees a maximum worst case execution time for each access sequence. Low latency requests are given precedence as with the Bank Scheduler.

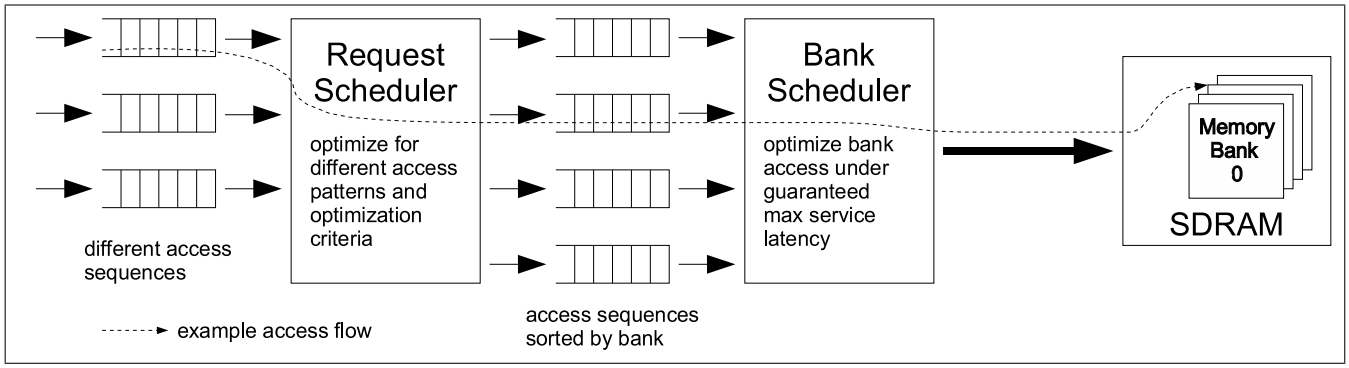

Figure 2: SDRAM controller architecture

3.3 FIFOs

Two FIFO stages exist in this design. The first stage is the Request Buffer in which requests get buffered before they enter the Request Scheduler. The second stage is the Bank Buffer between the Request Scheduler and the Bank Scheduler, where the requests are stored sorted by bank. All FIFOs in both stages are relatively short with about 5 entries. Again, requests with a higher stream priority are allowed to "overtake" normal requests to provide short latencies.

3.4 Data flow

The write-request data first gets stored inside the Data Buffers. For later access, the according request gets a tag assigned. Once the request has been scheduled, the data is selected by that tag and transferred to the SDRAM. Read requests work the opposite way.

3.5 Parameters

The controller is parameterizable to serve different applications. The most important parameters are the SDRAM layout (rows, columns, banks (n_{banks})) and timing (Table 1), number of inputs n_{input} and the FIFO sizes. Those parameters directly influence the latency and throughput. Generally, the maximum latency increases with increasing value of any parameter except for the stream priorities. The increase of n_s, n_{mc} and the FIFO sizes has greater impact than the SDRAM timing parameters due to the increasing round-robin cycle in the scheduler and the longer FIFO queues. Increasing n_b also increases the round robin cycle of the Bank Scheduler and thus the maximum latency, however, due to the request scattering over all banks the average latency increases less than linear. The introduction of stream priorities reduces the maximum latency for high priority streams, however the latency for normal streams increases. The maximum throughput depends on n_b, n_{mc} and the FIFO sizes. Raising n_b up to n_{bf} increases the throughput strongly because up to this value not enough banks are available for full bank interleaving. From this value on the throughput still increases slowly, because the probability that consecutive requests are mapped to the same bank decreases with more banks available. The effect of increasing n_b is strong for small values with a smaller influence for higher values. FIFO sizes have the same effect, since increasing the FIFOs first gives the scheduler a bigger choice to select requests for different banks, but this effect depreciates with deeper FIFOs. Stream priorities have a very slight negative impact, since they might cause a higher bus direction switching activity.

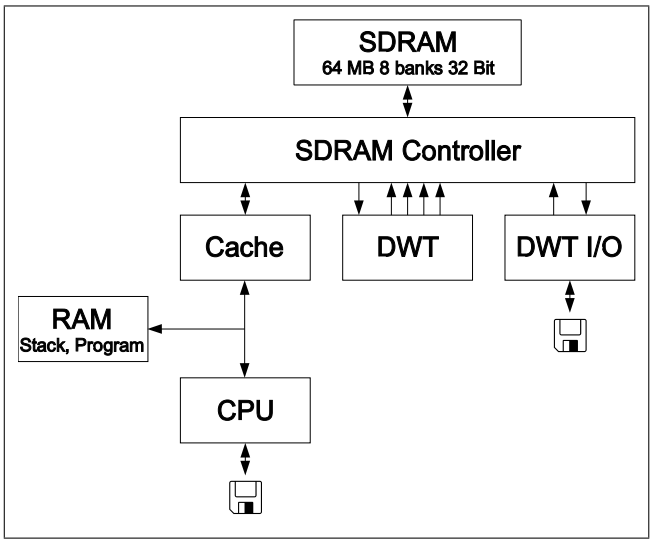

Figure 3: Simulator

4. SIMULATOR

We created an experimental system model for evaluation consisting of the SDRAM, SDRAM controller and two clients: a CPU with caches and a hardware implementation of a discrete wavelet transformation (DWT) algorithm (Figure 3).

We used *SystemC* for implementation because it allowed us to model at several abstraction levels in one language and to easily reuse available C models.

4.1 SDRAM controller and SDRAM model

For speed reasons, the SDRAM controller and the SDRAM model were implemented at transaction level, except for the interface between controller and SDRAM which is at RTL. To get meaningful results, clock synchronization points were inserted at several points and assumed latencies, for the function units derived from synthesizing all or part of these blocks, were added for every module.

4.2 Discrete Wavelet Transformation

The DWT implementation is based on [9] and was implemented in *SystemC* at RT-Level. The reason for doing it at RTL was that we needed a good idea about the speed of an FPGA implementation. After synthesis, place and route for a Xilinx $VirtexII$ ®

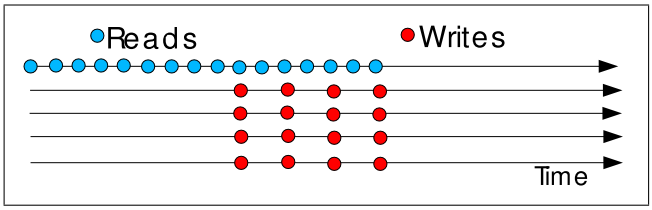

Figure 4: DWT data consumption/production timing example for a 32 pixel width image line

FPGA, the CAD tools reported a maximum operating frequency of 100 MHz. This frequency is not a integer fraction of the expected 150 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 FIFOs where build using special dedicated memory blocks inside the FPGA. The minimum size of one of these blocks is large enough to store half of one line (assuming an image width of 2048 pixels). Our DWT consumes two pixels per clock cycle and produces four periodic output streams with burst property, as seen in Figure 4. The period of the output streams depends on the image width and on the current DWT level [9]. 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 combination makes it possible to transform hard real-time latency constraints into soft real-time data-rate constraints. In Section 5 we will show how this transformation plays a key role in the overall system performance.

4.3 CPU

For CPU simulation, the SimpleScalar SimSafe DLX simulator together with an adaptation of the Dinero cache simulator were used. In reality, we would use a DSP with higher performance but we could not find an easily portable simulator code for it. To be able to use this CPU simulator inside the *SystemC* environment, we replaced the SimpleScalar memory model and created a "shell" *SystemC* module for the procedural C-code which provided the needed interfaces. To reflect the proposed flexible platform for image processing (Figure 1), the CPU was equipped with a SRAM for program and local variables (Figure 3). To simplify the design and coding of applications, all program and stack accesses where routed to the local RAM, whereas accesses to the heap area were routed to the SDRAM. Thus, any access to memory space returned by malloc() and new() went to the SDRAM. All this was done in 2 days including understanding the SimpleScalar source code.

5. SIMULATION RESULTS

To evaluate the concept, we did several tests. For the first experiment, the DWT was setup for a three level 512 x 512 grayscale image, and the CPU did a compression of a 128 x 128 color image using cjpeg from the media testbench [8]. The SDRAM controller and the DWT were clocked at 100 MHz, which our tools indicated is reachable with our FPGA configuration. The CPU and the cache were clocked at 1 GHz to compensate for the weak DLX performance (as said in Subsection 4.3 ideally we would use a powerful DSP instead). The SDRAM was an 8 bank, 128 MB module with t_{rcd}, t_{cl}, t_{wr}, $t_{rp} = 2$, $t_{cbr} = 2078$ and $t_{rc} = 12$. Datapaths were set to 32 bits. We measured the execution time of the DWT and the CPU for several runs. Results are shown in Table 2.

First, we did some tests with various buffer sizes. As a result, we see that a small Request Buffer of just one entry is sufficient for the request scheduler. Since only one request per clock cycle of 9 inputs is forwarded to the bank buffers and the clients are continuously issuing new requests, there are always enough requests available for scheduling. However, decreasing the bank buffer to one shows a negative effect. In this case, request sequences from one input are stalled as soon as the request goes to the same bank buffer as a previous request from the same or from a different sequence. Deeper FIFOs avoids these stalls allowing subsequent requests being scheduled, which often go to another bank. This leads to a better bank buffer utilization and thus gives the bank scheduler a greater flexibility to select and schedule banks.

Next, we gave the CPU accesses a higher priority. This is possible, since the DWT uses a long pre-fetch that hides even longer latencies. By decreasing the CPU access latency, we observe a slight CPU speedup, with almost no DWT slowdown.

For the second experiment, we gave the real time accesses a higher priority. This can be used if timing constrains are very strict and the guaranteed latency of the normal requests would not fit. For this experiment, we removed the DWT and the CPU and created two streams with 50/50 read/write accesses to random addresses, one "best effort" (BEF) stream with a fixed load and one "real-time" (RT) stream. Then we increased the load of the RT stream and tested for the first data loss on the RT would not fit. For this experiment, we created two streams with 50/50 read/write accesses to random addresses, one "best effort" (BEF) stream with a fixed load and one "real-time" (RT) stream. Then we increased the load of the RT stream and tested for the first data loss on the RT stream. At this point, we recorded the current RT load, the yielded BEF load and the latencies for both RT and BEF streams. We did this test with several BEF loads and with and without higher priorities for the RT stream. The results are shown in Table 3.

At high BEF loads, the loss on the RT stream occurred later with priorities activated. Figure 5 and 6 show the throughput and latencies of test number 3 and 4 (Table 3) over the time. Without priorities, the RT throughput increases while the latency stays constant until a maximum total SDRAM throughput of $\sim 95\%$ is reached, then the first data loss occurs. If the RT load is further raised, the yielded BEF rate starts to drop since the available bandwidth is split evenly

between the two streams. Since the SDRAM scheduler now cannot fulfill all requests, the buffers are always full and thus the latency increases.

With priorities activated, the yielded BEF rate starts to drop quickly when the overall SDRAM throughput of $\sim 95\%$ is reached, while the RT rate continues to increase without data losses. Also, the BEF latency starts to increase quickly. The point of first data loss occurs later compared to the no-priority experiment, however not at a RT rate of $\sim 95\%$ as someone would expect. Even with priorities activated, some BEF requests will interfere with RT requests and thus affect the maximal possible throughput for the prioritized stream.

Req. buff. size	Bank buf. size	cpu pri	Cache size / assoc.	DWT Mcycles	CPU Mcycles / CPI
10	10	no	2K / 4	1.4	26.8 (3.42)
10	5	no	2K / 4	1.4	26.8 (3.42)
10	1	no	2K / 4	1.4	28.0 (3.57)
5	5	no	2K / 4	1.4	26.7 (3.42)
1	5	no	2K / 4	1.4	26.7 (3.42)
1	5	no	2K / 4	1.4	26.7 (3.42)
1	5	yes	2K / 4	1.41	25.7 (3.29)

Table 2: Experiment 1: Simulation setup and results

Nr.	load BEF [w/c]	RT priori-tizing	Throughput total [w/c]	Throughput max. RT [w/c]	Throughput yield BEF [w/c]	Latency RT [cycles]	Latency BEF [cycles]
1	0.2	no	0.96	0.8	0.16	11 / 10	11 / 11
2	0.2	yes	0.96	0.8	0.16	10 / 10	11 / 10
3	0.5	no	0.94	0.5	0.44	80 / 55	81 / 52
4	0.5	yes	0.95	0.8	0.15	44 / 31	328 / 245
5	0.8	no	0.94	0.22	0.72	57 / 51	58 / 51
6	0.8	yes	0.95	0.57	0.28	22 / 20	143 / 85

w/c - words per cycle
BEF - Best Effort
RT - Real-time

Table 3: Experiment 2: real-time vs. best effort

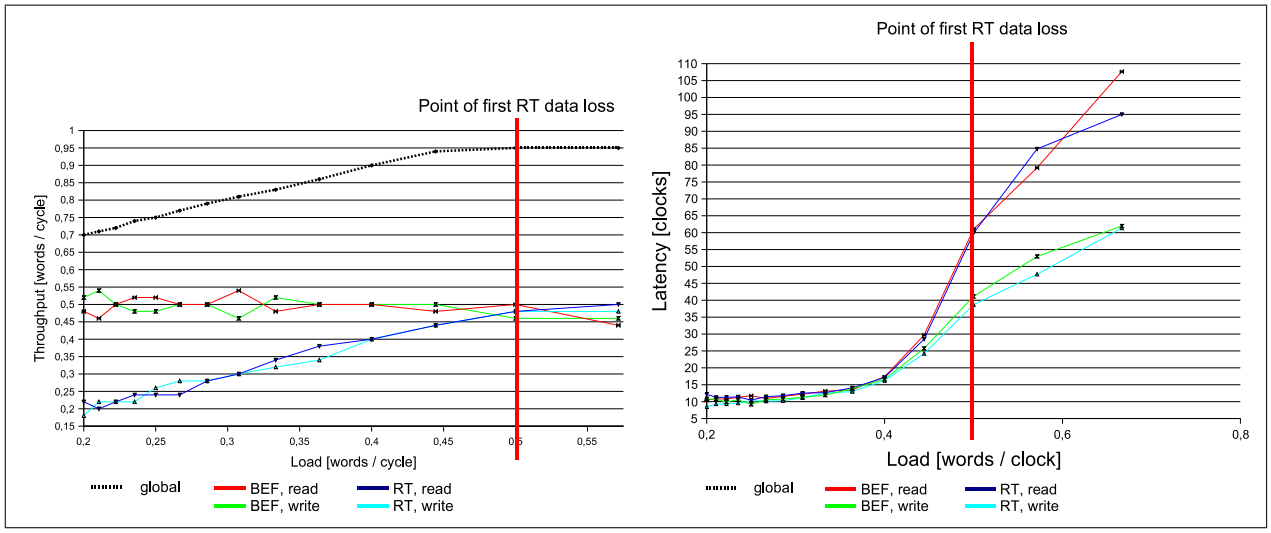

Figure 5: Experiment 2, real-time vs. best effort, test nr. 3

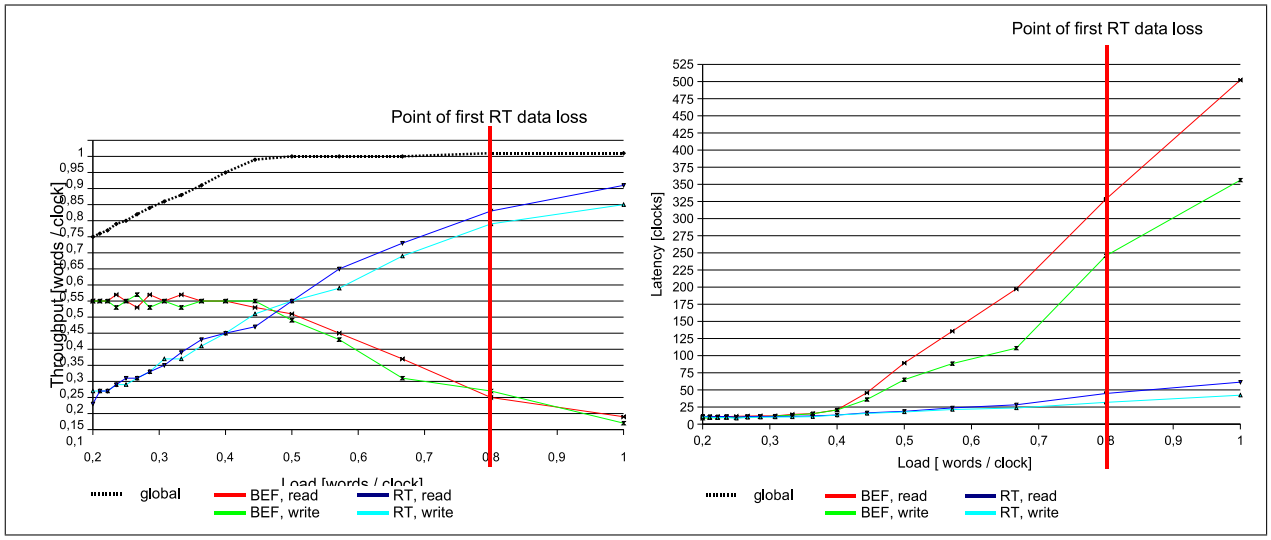

Figure 6: Experiment 2, real-time vs. best effort, test nr. 4

Although the current implementation of *SystemC* still showed a few pitfalls, the simulation performance and the flexibility were convincing. One test of experiment 1 above took about 5 minutes (gcc, 1.5GHz Athlon). A simple image pixel averaging scaling algorithm with no additional filtering, a simpler version of the SDRAM controller and a SDRAM model, thus overall a much less complex system, took about 480 minutes using Synopsys Scirocco on the same machine.

6. CONCLUSION

Based on earlier experience with a fixed architecture for a reconfigurable HDTV system, we presented 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 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.

7. REFERENCES

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

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

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

[4] Sonics siliconbackplane micronetwork overview. *http://www.sonicsinc.com/sonics/products/siliconbackplane*.

[5] Asheesh Khare, Preeti Ranjan Panda, N. D. A. N. High-level syntesis with syncronous and rambus drams. *IEICE E82-A*, 11 (1999).

[6] for blind review, O.

[7] Jeroen A. J. Leitjen, Jef L. van Meerbergen, A. H. T. J. A. G. J. Prophid: A heterogeneous multi-processor architecture for multimedia. *1997 International Conference on Computer Design (ICCD '97)* (October 1997), 164–169.

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

[9] 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).

[10] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P. R., and Owens, J. D. Memory access scheduling. In *ISCA* (2000), pp. 128–138.

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

[12] Weber, M. Arbiters: Design ideas and coding styles.

[13] 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.

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