# Methodology for Designing Statically Scheduled Application-Specific SDRAM Controllers using Constrained Local Search

Samuel Bayliss, George A. Constantinides

*Department of Electrical and Electronic Engineering, Imperial College London*
*London, UK*
s.bayliss08@imperial.ac.uk
g.constantinides@imperial.ac.uk

***Abstract*—This paper presents an automatic method for generating valid SDRAM command schedules which obey the timing restrictions of DDR2 memory from a set of memory references. These generated schedules can be implemented using a static memory controller. A complete knowledge of the sequence of memory references in an application enables the scheduling algorithm to reorder memory commands effectively to reduce latency and improve throughput. While statically scheduled command schedules might be considered too inflexible to be useful in mask-defined devices, they are well suited to implementation within an FPGA where new applications can be targeted by recompilation and reconfiguration. Static SDRAM schedules generated using our approach show a median 4× reduction in the number of memory stall cycles incurred across a selection of benchmarks when compared to schedules produced dynamically by the Altera High Performance Memory Controller.**

## I. Introduction

Much research effort has gone into producing EDA tools which raise the level of abstraction at which custom-hardware datapaths are designed [1]. By comparison, EDA tools largely ignore system-level issues outside the FPGA device and cannot infer interfaces to external memories. The approach to external dynamic memory interfacing using FPGAs differs very little from controllers employed in non-reconfigurable hardware. A controller IP block is instantiated which takes requests from the computation pipeline, dynamically handles the opening and closing of rows within the memory device and returns data alongside an *'acknowledge'* strobe some time later. The unpredictability of memory request latency using this approach means FPGA area is wasted introducing buffers to store incoming data, and stall signals must be incorporated into computation pipelines, which negatively affects achievable clock speeds. It should be noted that DRAM device behaviour *is* deterministic, and the latency of requests is predictable when the state of *'open'* rows in the memory device is considered. The controller wraps up this predictable behaviour in non-determinism because its generally assumed that memory access patterns are not known in advance, and that the request-acknowledge behaviour makes the controller IP block easy-to-use.

Given the effort taken to customize datapaths for FPGA designs, it seems quite plausible that realisable memory bandwidth can be increased and overall execution latency reduced by optimizing our memory requests for the deterministic behaviour of the SDRAM device. This paper shows advantages gained in predictability and memory perfomance by automatically generating a dynamic memory controller within an FPGA customized to a particular specific application.

Much prior work already exists on evaluating dynamic scheduling policies for SDRAM controllers to reduce average latency across benchmark suites, and some seeks to tune policies for specific applications. Our approach differs from these in that we describe a set of compile-time tools for constructing a viable static schedule for memory commands which meets the particular timing constraints of SDRAM devices. That schedule can be stored in a compact manner within the configuration bitstream of the FPGA and 'replayed' to efficiently fetch/store data required by a particular algorithm.

The key contributions described by this paper are:

- A local search algorithm which finds schedules that meet the interface constraints.
- A compact representation of this schedule within the FPGA device.
- A comparison of the efficiency of the approach against a conventional IP based approach.

## II. Background

Memory requests to SDRAM devices have specific timing constraints which arise from the design of the devices to achieve high density and low-pin count. SDRAM devices have multiple independent banks (typically four or eight per device). Commands are issued to each bank using a shared command interface. Each bank has two stable states, *idle* and *active*. An *'activate'* command must be issued for a particular row in a specific bank to transition the bank from the *idle* state to the *active* state before any read or write commands can be issued to the row. A *'precharge'* command must be issued to a bank to transition it to the *idle* state before

TABLE I
TYPICAL CONSTRAINTS WITHIN A DDR2 SDRAM DEVICE.

| Constraint | Time | Clks @ 266Mhz | Type | Meaning |
|---|---|---|---|---|
| $T_{rrd}$ | 10ns | 3 | global | minimum delay between any two activate commands to different banks. |
| $T_{rc}$ | 55ns | 15 | local | minimum delay between any two activate commands to the same bank. |
| $T_{rcd}$ | 15ns | 4 | local | minimum delay between activating a row and reading from it. |
| $T_{rtp}$ | 7.5ns | 2 | local | minimum delay between reading from a row and precharging it. |
| $T_{rp}$ | 15ns | 4 | local | minimum delay between precharging a row and activating it. |
| $T_{ras}$ | 40ns | 11 | local | minimum delay between activating a row and reading from it. |
| $T_{rr}$ | - | Burst-Length/2 | global | minimum delay between a read command and any other read command. |

activating a different row. Constraints on the sequencing of commands can be divided into two groups, local constraints which restrict command ordering within banks and global constraints which constrain the sequencing of commands to all banks. A subset of the constraints for a typical DDR2 device are shown in Table I. The static schedules developed by our approach must meet all of these constraints to ensure data is not corrupted.

Buffering frequently used data on-chip reduces the energy expended fetching data from external memory. In this work, the execution pipeline within the FPGA is decoupled from external memory using a scratch-pad memory implemented in dual-port RAM. Appropriate mappings of the memory requests from the execution pipeline to locations in the scratchpad memory and reverse mappings to external memory are discussed in [2], [3]. Modular address mappings from an external static memory into an on-chip memory buffer are discussed in [2]. In their work, all the data referenced by their target algorithm is loaded into the scratchpad memory before any execution begins. This is in contrast with our work which targets dynamic memory and where there is no specific loading phase. Instead the memory schedule is developed to allow execution to begin before the complete data set is loaded from external memory, *i.e.* the pipeline can begin consuming data as soon as the first data it needs are available, reducing the overall latency of the workload.

## III. RELATED WORK

There is a large body of work comparing different DRAM scheduling policies and controller designs. Most propose dynamic controllers and try to combine flexibility in unknown memory reference sequences with a limited memory of past events. Different fixed policies for scheduling DRAM commands are discussed in [4], its results show that none of the fixed policies they consider are optimal across all benchmarks. A dynamic policy for ordering DRAM commands is proposed in [5], but because the policy is implemented on-line, the system must make accurate predictions of future workload in order to select an appropriate response. These can only be based on a finite knowledge of past operations and the assumption that future operations follow similar execution patterns. Our static scheduling approach addresses problems with fixed and dynamic scheduling policies, since prior high-level knowledge of the algorithm being executed gives us insight into how the schedule might be optimized, and by explicitly stating the constrained command schedule, we avoid inefficiencies inherent in a rigid fixed policy scheduling. Static schedules are mentioned in [6] but dismissed as too inflexible to be useful in the SoC architecture they propose. While it may be difficult to provide flexible enough static scheduling in the context of a mask-implemented SoC (which must sell into a broad range of markets to justify an initial investment), FPGAs provide a generic platform upon which application specific static SDRAM controllers can be implemented directly in a reconfigurable fabric. Because this makes it possible to implement very specific controller designs meeting tight performance constraints without incurring mask costs, we feel a reexamination of the potential of statically scheduled SDRAM controllers in the context of FPGAs is warranted.

Constraint Logic Programming (CLP) [7] combines logic programming methods with domain specific constraint solvers. An overview of CLP applications in EDA tools can be found in [8]. Specific applications of CLP include the generation of self-test sequences [9] and its use in resource allocation [10] and scheduling [11] in high level synthesis using top-down search and branch and bound techniques. Since many search problems are too hard to solve with exact techniques, because the number of variables is too high or the structure of the problem hinders pruning of the search tree, non-deterministic local search techniques are often used, which start from an initial solution and through a series of moves, iterate towards a final solution.

## IV. LOCAL SEARCH PROCEDURE

The local search procedure produces a set of commands $P$. Each command has an associated $Type = \{Read, Write, Activate, Precharge\}$ given by the function $f_{type} : P \rightarrow Type$. Each command in $P$ has a mapping to time ( $f_{time} : P \rightarrow \mathbb{N}$) such that $f_{time}$ is injective giving a total ordering over the schedule. The local search procedure seeks to find an optimal mapping of the commands to time ($f_{time}$) such that the overall latency of the schedule is minimized subject to the timing constraints given in Table I.

The input to our search procedure is a collection of memory references. Each memory reference has an execution time associated with it. Accesses to DDR2 SDRAM devices are made in bursts of four or eight words, hence the memory references are sorted into address order and packed into read commands. Each read command has a mapping to execution time ($f_{exec-time} : P \rightarrow \mathbb{N}$), derived from the earliest memory

reference packed within it. This indicates the time that data fetched must be ready on-chip for use in the execution pipeline. Starting from an initial schedule based on a read command ordering inferred by the execution time, a sequence of *'move'* and *'pack'* commands transforms the schedule whilst maintaining feasibility.

A *'move'* operation alters the ordering of read commands issued to one bank in the memory device. The ordering of the subset of read commands ($R_B \subset P$) to each bank ($B$) can be expressed by a relation on $R_B$. Let $Successor \subset P \times P$ be the transitive reduction of the relation $\{(c_1, c_2) | f_{time}(c_2) > f_{time}(c_1)\}$. This relation when applied to the subset of read commands which make up a bank ($R_B$) gives the ordering of read commands within that bank. An example of a move operation can be seen in Figure 1.

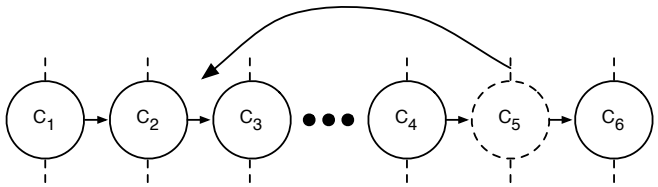


Fig. 1. Move Operation

Before the move:

$$Successor_{read}(C_1, C_2) \wedge Successor_{read}(C_2, C_3) \wedge Successor_{read}(C_4, C_5) \wedge Successor_{read}(C_5, C_6),$$

and afterwards,

$$Successor_{read}(C_1, C_2) \wedge Successor_{read}(C_2, C_5) \wedge Successor_{read}(C_5, C_3) \wedge Successor_{read}(C_4, C_6).$$

In essence, the move operation moves a command from one position in the ordering to another, producing a new total ordering of reads. Each move is selected to reduce the number of row swap operations (which introduce *'activate'* and *'precharge'* commands) in the schedule by clustering together read commands to the same row.

The *'pack'* operation is a procedure which interleaves commands across all banks according to a heuristic based on the slack available in the schedule. Given a subset ($S$) of the commands in $P$, where $S$ are all within the same memory bank, we define the function $f_{totalslack} : 2^P \rightarrow \mathbb{N}$ as :

$$f_{totalslack}(S) = \sum_{C \in S} min(0, slack(C))$$
$$where\ slack(C) = f_{exec-time}(C) - f_{time}(C) - \sum_{(D \in P | f_{time}(D) < f_{time}(C))} min(0, slack(D))$$

The pack procedure maintains a set of scheduled instructions ($Sched_B$) for each bank ($b$) and proceeds as follows.

**set** $Sched_b = \emptyset$ for all banks $b = 1, 2, \ldots, B$.
**while** $\bigcup_1^B Sched_b \neq P$ **do**

1) Evaluate $f_{totalslack}$ for the set of commands $S$ ($P \setminus Sched_b$) for each bank $b$.
2) Select a bank where $f_{totalslack}$ is most negative ($B_{sel}$).
3) Select the latest scheduled command $C_{pre}$ from $Sched_{B_{sel}}$.
4) Find *successor*$(C_{pre}, C_{succ})$ in $P$.
5) Schedule $C_{succ}$, i.e. alter $f_{time}(C_{succ})$ to move $C_{succ}$ as close in time to $C_{pre}$ without:
   - Scheduling $C_{succ}$ at the same time as any other command.
   - Violating local intra-bank ordering constraints.
   - Violating global activate-activate constraints.
6) $Sched_{B_{sel}} = Sched_{B_{sel}} \cup C_{succ}$.

**end while**

In our implementation, the constraint solver ensures that the conditions in step 5) are not violated. A series of alternating *'pack'* and *'move'* operations reduces the number of row swap operations and interleaves accesses to different banks.

## V. Schedule Implementation

After producing a valid constrained command schedule at compile time, appropriate hardware can be created to generate the schedule within the FPGA. Scheduled commands are represented in a 24-bit format with a 3-bit opcode, a 3-bit field to determine to which bank the command should be issued, a 14-bit field to hold row/column addresses and a 4-bit field to indicate a 0 to 15 cycle delay before issuing subsequent commands. In this format, two 18-Kbit block RAMs can hold 1536 commands (shown in Figure 2), which can accomodate a schedule of variable length between 1536 and 24576 cycles. The schedule can be extended as appropriate with the addition of further block RAMs.

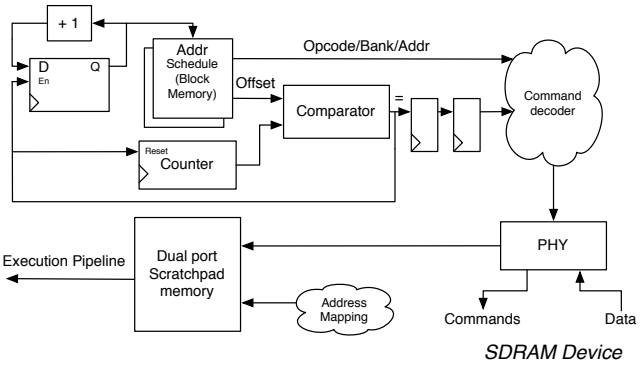


Fig. 2. Proposed static controller hardware

## VI. Results

Execution schedules and their associated memory references were generated from several representative benchmarks. A Sobel edge-detector defined as an affine loop body, a Radix-4 FFT code defined recursively and a matrix-matrix multiplication are considered. The IP-block based approach was evaluated using the Altera High Performance Memory Controller V9.0 which is interfaced using a four entry command FIFO buffer. The SDRAM device simulated has a 200Mhz command clock and 8-bit word size. Two different controllers were implemented to create comparison points with our statically scheduled approach. In the first, memory

TABLE II
SIMULATION RESULTS

| Benchmark | Stall Cycles (Dynamic without prefetch) | Stall Cycles (Dynamic with prefetch) | Negative Slack (Static) |
|---|---|---|---|
| 8x8 64-bit Matrix-Matrix Multiply | 1013 | 123 | 31 |
| 16x16 64-bit Matrix-Matrix Multiply (Row-Col) | 3818 | 195 | 131 |
| 16x16 64-bit Matrix-Matrix Multiply (Col-Row) | 4216 | 195 | 35 |
| 256-sample 32-bit Radix-4 FFT | 847 | 654 | 19 |
| 16x16 16-bit Sobel Edge Detect | 548 | 52 | 26 |

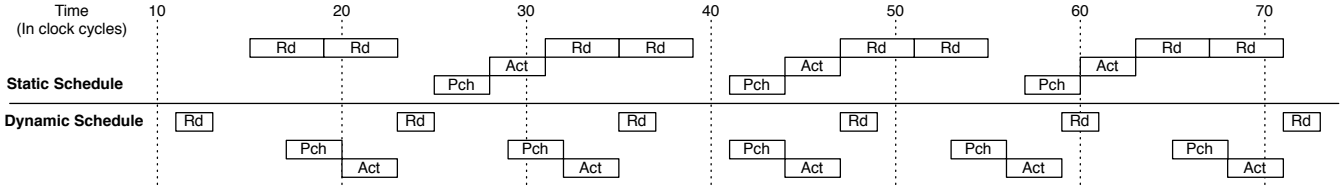


Fig. 3. Figure showing static and dynamic scheduling of FFT256 benchmark.

references were only sent to the controller 'on-demand' when they were first seen in the execution schedule. In the second controller, data was aggressively prefetched, keeping the DDR controller FIFO buffer full. In both cases, when data for the execution pipeline was unavailable, the controller would stall until data was returned from the controller. The number of stall cycles introduced due to unavailability of data is recorded in Table II. As would be expected, the prefetching approach hides much of the latency of the memory requests. A static schedule was generated alongside this and the number of stall cycles introduced by negative slack in the generated schedule is presented for comparison.

Table II shows that our static scheduled approach achieves a median reduction in read stall cycles of 4× across the selected benchmarks. Figure 3 shows the first 75 cycles of the FFT256 benchmark extracted from simulation of the dynamic controller and the output from the local search procedure. The static approach has selected bursts of eight beats while the dynamic controller only produces bursts of four beats, however since there are substantial gaps in the dynamic controller schedule and there is no increase in overall throughput by selecting longer bursts, this discrepancy is unlikely to contribute significantly to the improved performance of our schedule. Instead we can look at the increase in the average number of words fetched per row-swap. In Figure 3, the static schedule loads 64 words in 3 row-swaps whilst the dynamic controller loads 24 words in 5 row-swaps. The local search algorithm has found a schedule which exploits locality to improve the throughput of the memory controller in a way that the dynamic controller was unable to do.

## VII. CONCLUSION

These results show an average 4× reduction in memory stalls across the presented benchmarks by statically scheduling the SDRAM accesses feeding a scratchpad memory buffer rather than using a dynamically scheduled SDRAM Controller IP.

## ACKNOWLEDGMENT

This work was supported by a UK EPSRC Doctoral Training Award.

## REFERENCES

[1] S. Edwards, "The challenges of synthesizing hardware from c-like languages," *Design & Test of Computers, IEEE*, vol. 23, no. 5, pp. 375–386, May 2006.

[2] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic on-chip memory minimization for data reuse," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, 2007, pp. 251–260.

[3] E. De Greef, F. Catthoor, and H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications," *Parallel Computing*, vol. 23, no. 12, pp. 1811–1837, 1997.

[4] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 128–138, 2000.

[5] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 35th International Symposium on Computer Architecture*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 39–50.

[6] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable sdram memory controller," in *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2007, pp. 251–256.

[7] K. Apt, *Principles of Constraint Programming*. Cambridge University Press, August 2003.

[8] R. Beckmann, U. Bieker, and I. Markhof, "Application of constraint logic programming for VLSI CAD tools," in *CCL '94: Proceedings of the First International Conference on Constraints in Computational Logics*. London, UK: Springer-Verlag, 1994, pp. 183–200.

[9] U. Bieker and P. Marwedel, "Retargetable self-test program generation using constraint logic programming," in *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*. New York, NY, USA: ACM, 1995, pp. 605–611.

[10] K. Kuchcinski, "An approach to high-level synthesis using constraint logic programming," in *Euromicro Conference, 1998. Proceedings. 24th*, vol. 1, Aug 1998, pp. 74–82 vol.1.

[11] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 3, pp. 355–383, 2003.