

Memory aware compilation through accurate timing extraction*

Appeared in DAC 2000

Peter Grun
pgrun@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Alex Nicolau
nicolau@cecs.uci.edu

Center for Embedded Computer Systems
University of California, Irvine, CA 92697-3425, USA

Abstract

Memory delays represent a major bottleneck in embedded systems performance. Newer memory modules exhibiting efficient access modes (e.g., page-, burst-mode) partly alleviate this bottleneck. However, such features can not be efficiently exploited in processor-based embedded systems without memory-aware compiler support. We describe a memory-aware compiler approach that exploits such efficient memory access modes by extracting accurate timing information, allowing the compiler's scheduler to perform global code reordering to better hide the latency of memory operations. Our memory-aware compiler scheduled several benchmarks on the TI C6201 processor architecture interfaced with a 2-bank synchronous DRAM and generated average improvements of ~24% over the best possible schedule using a traditional (memory-transparent) optimizing compiler, demonstrating the utility of our memory-aware compilation approach.

1 Introduction

In recent SOC and processor architectures, memory is identified as a key performance and power bottleneck [19]. With advances in memory technology, new memory modules (e.g., SDRAM, DDRAM, RAMBUS, etc.) that exhibit efficient access modes (such as page mode, burst mode, pipelined access [18], [19]) appear on the market with increasing frequency. However, without compiler support, such novel features cannot be efficiently used in a programmable system.

Furthermore, in the context of Design Space Exploration (DSE), the system designer would like to evaluate different combinations of memory modules and processor cores from an IP library, with the goal of optimizing the system for varying goals such as cost, performance, power, etc.. In order to take full advantage of the features available in each such processor-memory architecture configuration, the compiler needs to be aware of the characteristics of each memory library component.

Whereas optimizing compilers have traditionally been designed to exploit special architectural features of the processor (e.g., detailed pipelining information), to our knowledge no prior work has addressed memory-library-aware compilation tools that explicitly model and exploit the high-performance features of such diverse memory modules. Indeed, particularly for memory modules, a more accurate timing model for the different memory

*This work was partially supported by grants from NSF (MIP-9708067) and ONR (N00014-93-1-1348).

access modes allows for a better match between the compiler and the memory sub-system, leading to better performance.

Traditionally, these access modes were transparent to the processor, and were exploited implicitly by the memory controller (e.g., whenever a memory access referenced an element already in the DRAM's row buffer, it avoided the row-decode step, fetching it directly from the row buffer). However, the memory controller only has access to local information, and is unable to perform more global optimizations (such as global code reordering to better exploit special memory access modes). By providing the compiler with a more accurate timing model for the specific memory access modes, it can perform global optimizations that effectively hide the latency of the memory operations, and thereby generate better performance.

In this paper we present such an approach that allows the compiler to exploit detailed timing information about the memory modules, providing an opportunity to perform global optimizations. The key idea is that we combine the timing model of the memory modules (e.g., efficient memory access modes) with the processor pipeline timings to generate accurate operation timings. We then use these exact operation timings to better schedule the application, and hide the latency of the memory operations.

In Section 2 we present previous work addressing memory and special access mode related optimizations. In Section 3 we show a simple motivating example to illustrate the capabilities of our technique, and in Section 4 we present the flow of our approach. In Section 5 we describe our timing generation algorithm. In Section 6 we present a set of experiments that demonstrate the use of accurate timing for the TIC6201 architecture with a synchronous DRAM module, and present the performance improvement obtained by our memory-aware compiler. We conclude with a summary in Section 7.

2 Related Work

Related work on memory optimizations has been addressed previously both in the custom hardware synthesis and in the embedded processor domains. In the context of custom hardware synthesis, several approaches have been used to model and exploit memory access modes. Ly et. al. [15] use behavioral templates to model complex operations (such as memory reads and writes) in a CDFG, by enclosing multiple CDFG nodes and fixing their relative schedules (e.g., data is asserted one cycle after address for a memory write operation). However, since different memory operations are treated independently, the scheduler has to assume a worst-case delay for memory operations – missing the opportunity to exploit efficient memory access modes such as page and burst mode.

Panda et. al. [17] outline a pre-synthesis approach to exploit efficient memory access modes, by massaging the input application (e.g., loop unrolling, code reordering) to better match the behavior to a DRAM memory architecture exhibiting page-mode accesses. Khare et. al. [12] extend this work to Synchronous and

RAMBUS DRAMs, using burst-mode accesses, and exploiting memory bank interleaving.

Wuytack et. al. [22] present an approach to increase memory port utilization, by optimizing the memory mapping and code re-ordering. Our technique complements this work by exploiting the efficient memory access modes to further increase the memory bandwidth.

While the above techniques work well for custom hardware synthesis, they cannot be applied directly to embedded processors containing deep pipelines and wide parallelism: a compiler must combine the detailed memory timing with the processor pipeline timing to create the timing for full-fledged operations in the processor-memory system.

Processors traditionally rely on a memory controller to synchronize and utilize specific access modes of memory modules (e.g., freeze the pipeline when a long delay from a memory read is encountered). However, the memory controller only has a local view of the (already scheduled) code being executed. In the absence of an accurate timing model, the best the compiler can do is to schedule optimistically, assuming the fastest access time (e.g., page mode), and rely on the memory controller to account for longer delays, often resulting in performance penalty. This optimistic approach can be significantly improved by integrating an accurate timing model into the compiler. In our approach, we provide a detailed memory timing model to the compiler so that it can better utilize efficient access modes through global code analysis and optimizations, and help the memory subsystem produce even better performance. We use these accurate operation timings in our retargetable compiler to better hide the latency of the memory operations, and obtain further performance improvements.

Moreover, in the absence of dynamic data hazard detection (e.g., in VLIW processors), these operation timings are *required* to insure correct behavior: the compiler uses them to insert NOPs in the schedule to avoid data hazards. In the absence of a detailed timing model, the compiler is forced to use a pessimistic schedule, thus degrading overall performance.

In the embedded and general purpose processor domain, a new trend of instruction set modifications has emerged, targeting explicit control of the memory hierarchy, through for instance prefetch, cache freeze, and evict-block operations (e.g., TriMedia 1100, StrongArm 1500, IDT R4650, Intel IA 64, Sun UltraSPARC III, etc. [1]). Even though such operations can improve memory traffic behavior, they are orthogonal to the specific library modules used. By exploiting detailed timing of efficient memory access modes, we can provide further performance improvement opportunities.

In the domain of programmable SOC architectural exploration, recently several efforts have proposed the use of Architecture Description Languages (ADLs) to drive generation of the software toolchain (compilers, simulators, etc.) ([4], [5], [8], [9], [14]). However, most of these approaches have focused primarily on the processor and employ a generic model of the memory subsystem. For instance, in the Trimaran compiler [21], the scheduler uses operation timings specified on a per-operation basis in the MDes ADL to better schedule the applications. However they use fixed operation timings, and do not exploit efficient memory access modes. Our approach uses EXPRESSION [9], a memory-aware ADL that explicitly provides a detailed memory timing to the compiler and simulator.

Moreover, in the context of Design Space Exploration (DSE), when different processor cores and memory IP modules are mixed and matched to optimize the system for different goals, describing the timings on a per-operation basis requires re-specification whenever the pipeline architecture or the memory module is

changed. Since changes in the pipeline and memory architecture impact the timings of all operations (not only memory related), updating the specification during every DSE iteration is a very cumbersome and error prone task. Instead, in our approach we extract these operation timings from the processor core pipeline timing and the storage elements timings, allowing for faster DSE iterations. To our knowledge no previous compiler can exploit detailed timing information of efficient memory access modes offered by modern DRAM libraries (e.g., SDRAM, RAMBUS).

Recent work on interface synthesis [2], [3] present techniques to formally derive node clusters from interface timing diagrams. These techniques can be applied to provide an abstraction of the memory module timings required by our algorithm.

Our technique generates accurate operation timing information by marrying the pipeline timing information from the processor core module, with timing information from the memory library modules, and uses it in a parallelizing compiler to better target the processor-memory system. We exploit efficient memory access modes such as page-mode and burst-mode, to increase the memory bandwidth, and we use the accurate timing information to better hide the latency of the memory operations, and create significant performance improvements. We support fast DSE iterations, by allowing the designer to plug in a memory module from an IP library, and generate the operation timings, in a compiler-usable form.

3 Motivating Example

A typical efficient access mode for contemporary DRAMs (e.g., SDRAM) is burst mode access, that is not fully exploited by traditional compilers. We use a simple example to motivate the performance improvement made possible by compiler exploitation of such access modes through a more accurate timing model.

The sample memory library module we use here is the IBM0316409C [11] Synchronous DRAM. This memory contains 2 banks, organized as arrays of 2048 rows x 1024 columns, and supports normal, page mode, and burst mode accesses. A normal read access starts by a row decode (activate) stage, where the entire selected row is copied into the row buffer. During column decode, the column address is used to select a particular element from the row buffer, and output it. The normal read operation ends with a precharge (or deactivate) stage, wherein the data lines are restored to their original values.

For page mode reads, if the next access is to the same row, the row decode stage can be omitted, and the element can be fetched directly from the row buffer, leading to a significant performance gain. Before accessing another row, the current row needs to be precharged.

During a burst mode read, starting from an initial address input, a number of words equal to the burst length are clocked out on consecutive cycles without having to send the addresses at each cycle.

Another architectural feature which leads to higher bandwidth in this DRAM is the presence of two banks. While one bank is bursting out data, the other can perform a row decode or precharge. Thus, by alternating between the two banks, the row decode and precharge times can be hidden. Traditionally, the architecture would rely on the memory controller to exploit the page/burst access modes, while the compiler would not use the detailed timing model. In our approach, we incorporate accurate timing information into the compiler, which allows the compiler to exploit more globally such parallelism, and better hide the latencies of the memory operations.

We use the simple example in Figure 1 (a) to demonstrate the performance of the system in three cases: (I) without efficient

access modes, (II) optimized for burst mode accesses, but without an accurate timing model, and (III) optimized for burst mode accesses with an accurate timing model.

The primitive access mode operations for a Synchronous DRAM are shown in Figure 1 (b): the un-shaded node represents the row decode operation (taking 2 cycles), the solid node represents the column decode (taking 1 cycle), and the shaded node represents the precharge operation (taking 2 cycles).

Figure 1 (c) shows the schedule for the unoptimized version, where all reads are normal memory accesses (composed of a row decode, column decode, and precharge). The dynamic cycle count here is $9 \times (5 \times 4) = 180$ cycles.

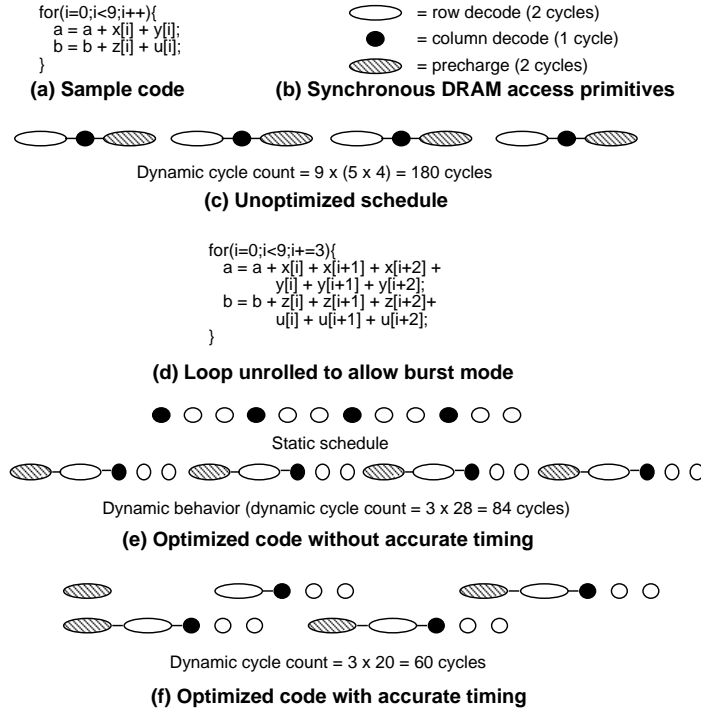


Figure 1. Motivating example

In order to increase the data locality and allow burst mode access to read consecutive data locations, an optimizing compiler would unroll the loop 3 times. Figure 1 (d) shows the unrolled code, and Figure 1 (e) shows the static schedule, and the dynamic (run-time) schedule of the code¹, for a schedule with no accurate timing. Traditionally, the memory controller would handle all the special access modes implicitly, and the compiler would schedule the code optimistically, assuming that each memory access takes 1 cycle (the length of a page mode access). During a memory access which takes longer than expected, the memory controller has to freeze the pipeline, to avoid data hazards. Thus, even though the static schedule seems faster, the dynamic cycle-count in this case is $3 \times 28 = 84$ cycles.

Figure 1 (f) shows the effect of scheduling using accurate memory timing on code that has already been optimized for burst mode. Since the memory controller does not need to insert stalls anymore, the dynamic schedule is the same as the static one (shown in Figure 1 (f)). Since accurate timing is available, the scheduler can hide the latency of the precharge and row decode stages, by for instance precharging at the same time the two banks, or executing row decode while the other bank bursts out data. The dynamic cycle count here is $3 \times 20 = 60$ cycles, resulting in a 40% improvement over the best schedule a traditional optimizing compiler would generate.

¹In Figure 1 (c) the static schedule and the run-time behavior were the same. Here, due to the stalls inserted by the memory controller, they are different

Thus, by providing the compiler with more detailed information, the efficient memory access modes can be better exploited. The more accurate timing model creates a significant performance improvement, in addition to the page/burst mode optimizations.

4 Our Approach

In our IP library based Design Space Exploration (DSE) scenario, the designer starts by selecting a set of components from a processor IP library, and a memory IP library. Our EXPRESSION [9] Architecture Description Language (ADL) (containing a mix of such IP components and custom blocks), is used to target the compilation process to the specific processor-memory architecture chosen by the designer. In order to have a good match between the compiler and the architecture, detailed information about resources, memory features, and timing has to be provided.

Furthermore, contemporary high-end embedded processors contain deep pipelines and wide parallelism, thus hazards in the pipeline can create performance degradation, or even incorrect behavior, if undetected. There are three types of hazards in the pipeline: resource, data, and control hazards [10]. In [7] we addressed the resource hazards problem by automatically generating the detailed Reservation Tables needed to detect structural hazards in the pipeline.

Data hazards occur when the pipeline changes the order of read/write accesses to operands, breaking data dependencies [10]. In this paper we address the data hazards problem, by marrying the timing information from memory IP modules with a structural description of the processor pipeline architecture, and generate the operation timings needed by the compiler to avoid such data hazards.

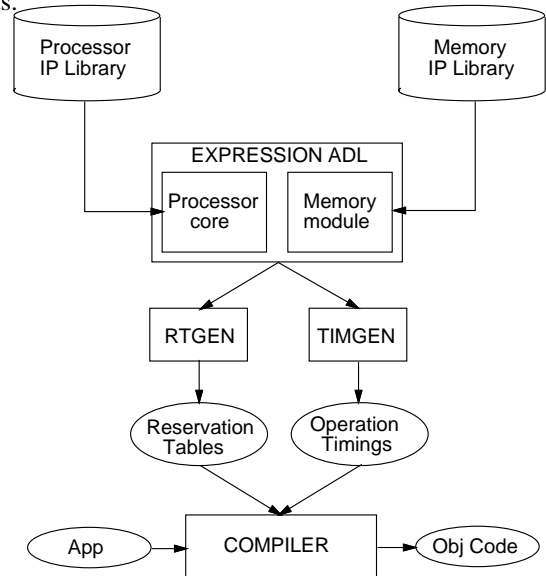


Figure 2. The Flow in our approach

Since memory is a major bottleneck in the performance of such embedded systems, we then show how this accurate timing model can be used by the compiler to better exploit memory features such as efficient access modes (e.g., page-mode and burst-mode), to obtain significant performance improvements.

Furthermore, we provide the system designer with the ability to evaluate and explore the use of different memory modules from the IP library, with the memory-aware compiler fully exploiting the special efficient access modes of each memory component.

5 Timing extraction algorithm

Timing of operations has been used for a long time to detect data hazards [10]. However, compilers traditionally used

fixed timings for scheduling operations. In the absence of dynamic (run-time) data hazard detection capabilities in the hardware (such as in VLIW processors), the compiler has to use conservative timings, to avoid such hazards, and ensure correct behavior. In the presence of dynamic control capabilities the compiler may alternatively use optimistic timings, and schedule speculatively, knowing that the hardware will account for the potential delay.

For memory-related operations, the memory controller provides some dynamic control capabilities, which can for instance stall the pipeline if a load takes a long time. In this case a traditional compiler would use fixed timings to schedule either conservatively or optimistically the memory operations. In the conservative approach, the compiler assumes that the memory contains no efficient access modes, and all accesses require the longest delay. Even though this results in correct code, it wastes a lot of performance. Alternatively, the optimistic schedule is the best the compiler could do in the absence of accurate timing: the compiler assumes that all the memory accesses have the length of the fastest access (e.g., page mode read), and relies on the memory controller to account for the longer delays (e.g., by freezing the pipeline). Clearly, this creates an improvement over the conservative schedule. However, even this optimistic approach can be significantly improved by integrating a more accurate timing model. In our approach, the compiler better exploits the efficient memory access modes, by better hiding the latency of the memory operations, and creating considerable further performance improvements.

Our timing generation algorithm, called TIMGEN, starts from an EXPRESSION ADL [9] containing a structural description of the processor pipeline architecture, and an abstraction of the memory module access mode timings, and generates the operation timings, consisting of the moments in time (relative to the issue cycle of the operation) when each operand in that operation is read (for sources) or written (for destinations).

To illustrate the TIMGEN algorithm, we use the example architecture in Figure 3, based on the TI TMS320C6201 VLIW DSP, with an SDRAM block attached to the External Memory Interface (EMIF). For the sake of illustration, Figure 3 presents the pipeline elements and the memory timings only for load operations executed on the D1 functional unit.

The TIC6201 processor contains 8 functional units, an 11-stage pipeline architecture, and no data cache. The pipeline contains: 4 fetch stages (PG, PS, PW, PR), 2 decode stages (DP, DC), followed by the 8 pipelined functional units (L1, M1, S1, D1, L2, M2, S2 and D2). In Figure 3 we present the D1 load/store functional unit pipeline with 5 stages: D1_E1, D1_E2, EMIF, MemCtrl_E1 and MemCtrl_E2. The DRAM1 represents the SDRAM memory module attached to the EMIF, and RFA represents one of the two TIC6201 distributed register files.

The D1_E1 stage computes the load/store address, D1_E2 transfers the address across the CPU boundary, to the EMIF, which sends the read/write request to the memory module. Since SDRAM access may incur additional delay depending on the memory access mode, the EMIF will wait until the SDRAM access is finished. MemCtrl_E1 then transfers the data received from EMIF back into the CPU, and MemCtrl_E2 writes it to the destination register.

The graphs inside the DRAM1 of Figure 3 are the SDRAM's memory timings represented as an abstraction of the memory access modes, through combinations of a set of primary operation templates corresponding to the row-decode, column-decode and precharge (similar to Figure 1 (b)). Normal Read (NR) represents a normal memory read, containing a row-decode, column-

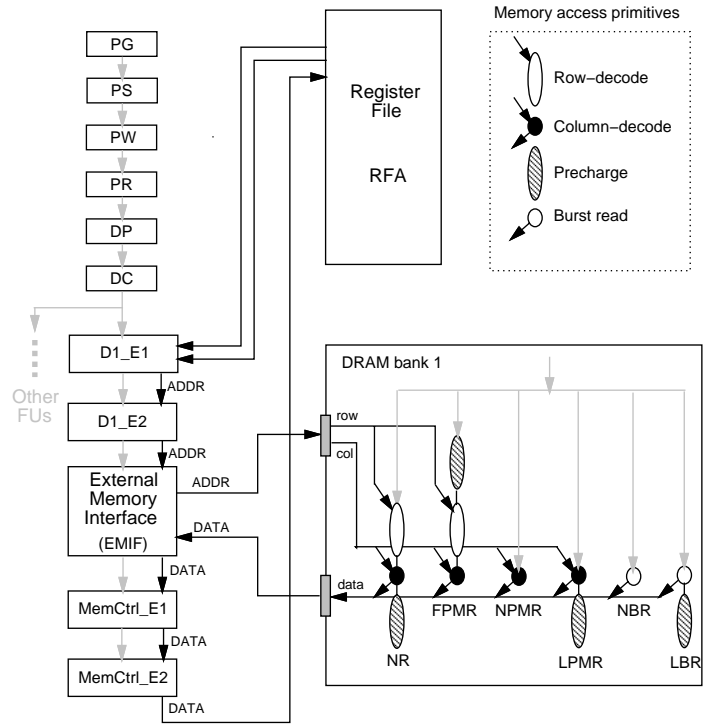


Figure 3. Example architecture, based on TI TMS320C6201

decode and precharge. First Page Mode (FPMR) represents the first read in a sequence of page-mode reads, which contains a row precharge, a row-decode and a column decode. The next page mode operations in the sequence, shown as Next Page Mode (NPMR), contain only a column-decode. Similarly, Next Burst (NBR) represents burst-mode reads which contain only the bursting out of the data from the row-buffer. The Last Page Mode (LPMR) and Last Burst Mode (LBR) Read are provided to allow precharging the bank at the moment when the page-mode or burst sequence ends (instead of waiting until the next sequence starts), thus allowing for more optimization opportunities.

The TIMGEN algorithm is outlined in Figure 4. The basic idea behind the TIMGEN algorithm is that during execution, an operation proceeds through a pipeline path, and accesses data through some data transfer paths. For memory operations, the data transfer paths can access complex memory modules (e.g., SDRAM), and the operation may be delayed based on the memory access mode (e.g., normal/page/burst mode). Therefore, we can collect detailed timing information regarding each operand for that operation, by tracing the progress of the operation through the pipeline and storage elements.

To demonstrate how the TIMGEN algorithm works, we use the load operation $LDW .D1 RFA0[RFA1], RFA2$, executed on the architecture shown in Figure 3. Assuming this load is a first read in a sequence of page-mode reads, we want to find in what cycle the base argument (RFA0), offset argument (RFA1), and the implicit memory block argument (the DRAM bank) are read, and in what cycle the destination register argument (RFA2) is written.

Detailed timing is computed by traversing the pipeline paths, data transfer paths, and complex memory modules, and collecting the cycle where each argument is read/written. In this instance, the $LDW .D1 RFA0[RFA1], RFA2$ operation starts by traversing the 4 fetch pipeline stages (PG, PS, PW, PR), and 2 decode stages (DP, DC). Then, the operation is directed to the stage 1 of the D1 functional unit (D1_E1)². Here, the two source operands, repre-

²Due to space considerations, the other functional units have not been represented

Algorithm: TIMGEN

Input: Processor pipeline and memory access mode timings description, a memory operation, and the access_mode used (normal/page/burst)

Output: Timings for all operation arguments

Begin TIMGEN

For curr_pipe_stage traversing the pipeline stages used by the input operation (in order of execution)

For all the data_transfer_paths accessed in the curr_pipeline_stage

Find the operation_argument corresponding to this data_transfer_path

Find the storage_element accessed by this data_transfer_path

If storage_element is a complex storage module (e.g., an SDRAM module)

Choose the access_pattern in this storage_element, according to the access_mode

Find the cycle when the operation_argument is transferred in this access_pattern

Add (cycle + curr_pipe_stage) as the timing for the operation_argument

Else If storage_element is a simple storage module (e.g., register file)

Add the curr_pipe_stage as the timing for the operation_argument

Update the curr_pipe_stage

End TIMGEN

Figure 4. The TIMGEN timing generation algorithm.

sending the base register and the offset register, are read from the RFA register file (RFA0 and RFA1), and the address is computed by left-shifting the offset by 2, and adding it to the base. Thus, the base and offset are read during the stage 7 of the pipeline.

During stage D1_E2, the address is propagated across the CPU boundaries, to the External Memory Interface (EMIF). Here, the EMIF sends the address to the SDRAM block, which starts reading the data. Depending on the type of memory access (normal/page/burst) and on the context information (e.g., whether the row is already in the row buffer), the SDRAM block may require a different amount of time. Assuming that this load operation is the first in a sequence of page mode reads, TIMGEN chooses the First Page Mode (FPM) template from the SDRAM block. Here we need to first precharge the bank, then perform a row-decode followed by a column decode. During the row-decode, the row part of the address is fed to the row decoder, and the row is copied into the row buffer. During column-decode, the column part of the address is fed to the column decoder, and the corresponding data is fetched from the row buffer. Since the prefetch and row-decode require 2 cycles each, and the column-decode requires 1 cycle, the data will be available for the EMIF 5 cycles after the EMIF sent the request to the SDRAM block. Thus the data is read 14 cycles after the operation was issued.

During the MemCtrl_E1 stage, the data is propagated back across the CPU boundary. The MemCtrl_E2 stage writes the data to the register file destination (RFA2), 16 cycles after the operation issue.

In this manner, the TIMGEN algorithm (Figure 4) computes the detailed timings of all the arguments for that given operation: the base argument (RFA0) and the offset argument (RFA1) are read during stage 7 of the pipeline, the memory block (the SDRAM array) is read during stage 14, and the destination argument (RFA2) is written during stage 16. A detailed description of the algorithm can be found in [6].

The worst case complexity of TIMGEN is $O(x * y)$, where x is the maximum number of pipeline stages in the architecture, and y the maximum number of data transfers in one pipeline stage. Typically, most architectures have between 5 and 15 stages in the pipeline, and 1 to 3 data transfer paths per pipeline stage, leading to reasonable computation time when applied to different pipeline architectures and memory modules with varying access modes. Since during compilation, the timings for all the operations in the application are required, we can compute before-hand these timings, and store them in a database.

6 Experiments

We now present a set of experiments demonstrating the performance gains obtained by using accurate timing in the compiler. We first optimize a set of benchmarks to better utilize the efficient memory access modes (e.g., through memory mapping, code re-ordering or loop unrolling³ [17]), and then we use the accurate timing model to further improve the performance by hiding the latencies of the memory operations. To separate out the benefit of the better timing model from the gain due to the access mode optimizations and the access modes themselves, we present the set of results which show the performance gains obtained by scheduling with accurate timing in the presence of a code already optimized for memory accesses, and compare them to the performance of the same memory-access-optimized code using less accurate timing, scheduled optimistically, assuming the shortest access time available (page-mode access), and relying on the memory controller to account for longer delays. This optimistic scheduling is the best alternative available to the compiler, short of an accurate timing model. We also compare the above approaches to the performance of the system in the absence of efficient memory access modes.

6.1 Experimental Setup

In our experiments, we use an architecture based on the Texas Instruments TMS320C6201 VLIW DSP, with one IBM0316409C synchronous DRAM [11] block exhibiting page-mode and burst-mode access, and 2 banks. The TIC6201 is an integer-point 8-way VLIW processor, with no data cache, as explained earlier. The External Memory Interface (EMIF) [20] allows the processor to program information regarding the memory modules attached, and control them through a set of control registers. We assume the SDRAM has the capability to precharge a specific memory bank (using the DEAC command), or both memory banks at the same time (using the DCAB command), and to perform a row decode while the other bank bursts out data (using the ACTV command).

The applications have been compiled using our EXPRESS re-targetable optimizing ILP compiler. Our scheduler, reads the accurate timing model generated by the TIMGEN algorithm, and uses Trailblazing Percolation Scheduling (TiPS) [16] to better target the specific architecture. TiPS is a powerful Instruction Level Parallelism (ILP) extraction technique, which can fully exploit the accurate operation timings, by highly optimizing the schedule. The cycle counts have been computed using our structural cycle-accurate simulator SIMPRESS [13].

6.2 Results

The first column in Table 1 shows the benchmarks we executed (from multimedia and DSP domains). The second column shows the dynamic cycle counts when executing the benchmarks on the architecture using only normal memory accesses (no efficient memory access modes, such as page/burst mode).

The third column represents the dynamic cycle counts optimized for efficient access modes, but with no accurate timing model. For fairness of the comparison we compile using the optimistic timing model, and relying on the memory controller to stall the pipeline when the memory accesses take longer than expected. Notice that by using these efficient access modes, and optimizing the code to better exploit them (e.g., by loop unrolling), we obtain a very high performance gain over the base-line un-optimized case.

The fourth column represents the dynamic cycle counts using both efficient access modes, and using an accurate timing model

³Loop unrolling trades-off performance against code size and register pressure [17] (we assumed 32 registers available).

to better hide the latency of the memory operations. By comparing these figures to the previous column, we separate out the performance gains obtained by using an accurate timing model, in addition to the gains due to the efficient access mode optimizations. The fifth column shows the percentage performance improvement of the results for the code optimized with accurate timing model (fourth column), compared to the results optimized without accurate timing model (third column).

The performance gains from exploiting detailed memory timing vary from 6% (in GSR, where there are few optimization opportunities), to 47.9% (in SOR, containing accesses which can be distributed to different banks and memory pages), and an average of 23.9% over a schedule that exploits the efficient access modes without detailed timing.

Benchmark	Unoptimized (normal accesses)	Optimized (page/burst mode)		
		Optimized code w/o accurate timing	Optimized code w/ accurate timing	% perf gain
SOR	192286	104350	70510	47.9
GSR	176034	111973	105346	6
beam	439087	41734	30854	35.2
idct	27163	8603	6939	23.9
mm	12909	3336	2568	29.9
dhrc	13957	6277	5637	10
madd	4005	987	727	35.7
dequant	6105	5105	4605	10
leaf_plus	2905	2405	1955	23
lowpass	7433	4169	3529	18
Average				23.9

Table 1. Dynamic cycle counts for the TIC6201 processor with an SDRAM block exhibiting 2 banks, page and burst accesses

The large performance gains obtained by using the more accurate operation timings are due to the better opportunities to hide the latency of the lengthy memory operations, by for instance performing row-decode, or precharge on one bank, while the other bank is bursting out data, or using normal compute operations to hide the latency of the memory operations. This effect is particularly large in the benchmarks where there are many operations available to hide in parallel with the memory operations (e.g., SOR, beam, IDCT, etc.). In cases when the memory access patterns do not allow efficient exploitation of the SDRAM banks (e.g., GSR), the gain is smaller. However, we are able to exploit accurate timing information for many multimedia and DSP applications, since they usually contain multiple arrays, often with two or more dimensions, which can be spread out over multiple memory banks.

7 Summary

We presented an approach which allows the compiler to exploit efficient memory access modes, such as page-mode and burst-mode, offered by modern DRAM families. We hide the latency of the memory operations, by marrying the timing information from the memory IP module with the processor pipeline timings, to generate accurate operation timings.

Traditionally, the memory controller accounted for memory delays, by freezing the pipeline whenever memory operations took longer than expected. However, the memory controller has only a local view of the code, and can perform only limited optimizations (e.g., when reading an element which is already in the row buffer, it avoids the column decode step). In the presence of efficient memory accesses, it is possible to better exploit such features, and better hide the latency of the memory operations, by providing the compiler with accurate timing information for the efficient memory accesses, and allowing it to perform more global optimizations on the input behavior.

By better exploiting the efficient memory access modes provided by modern DRAM families and better hiding of the memory operation latencies through the accurate timing information,

we obtained significant performance improvements. Furthermore, in the context of Design Space Exploration (DSE), our approach provides the system designer with the ability to evaluate and explore the use of different memory modules from the IP library, with the memory-aware compiler fully exploiting the efficient access modes of each memory component. We presented a set of experiments which separate out the gains due to the more accurate timing, from the gains due to the efficient access mode optimizations and access modes themselves. The average improvement was 23.9% over the schedule that exploits the access mode without detailed timing.

Currently our work applies to architectures without a data cache (e.g., the TI TMS320C6201, and many other DSP processors). However, note that we use the cache-like behavior of the DRAM row-buffer. Our on-going work investigates extensions to our algorithm to incorporate data caches and other embedded memory configurations.

8 Acknowledgments

We would like to acknowledge and thank Ashok Halambi, Nick Savoju, Asheesh Khare and Radu Cornea for their contributions to the EXPRESS/EXPRESSION project.

References

- [1] *HotChips conference, '97 - '99.*
- [2] P. Chou, R. Ortega, and G. Borriello. Interface co-synthesis techniques for embedded systems. In *ICCAD*, 1995.
- [3] K.-S. Chung, R. Gupta, and C. L. Liu. Interface co-synthesis techniques for embedded systems. In *ICCAD*, 1996.
- [4] G. H. et al. ISDL: An instruction set description language for retargetability. In *Proc. DAC*, 1997.
- [5] M. Freericks. The nML machine description formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.
- [6] P. Grun, N. Dutt, and A. Nicolau. Extracting accurate timing information to support memory-aware compilation. Technical report, University of California, Irvine, 1999.
- [7] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *ISSS*, San Jose, CA, 1999.
- [8] J. Gyllenhaal. A machine description language for compilation. Master's thesis, Dept. of EE, UIUC, IL., 1994.
- [9] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, Mar. 1999.
- [10] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers Inc, San Mateo, CA, 1990.
- [11] IBM Microelectronics, Data Sheets for Synchronous DRAM IBM0316409C. www.chips.ibm.com/products/memory/08J3348/.
- [12] A. Khare, P. R. Panda, N. D. Dutt, and A. Nicolau. High level synthesis with synchronous and rambus drams. In *SASIMI*, Japan, 1998.
- [13] A. Khare, N. Savoju, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proc. EUROMICRO*, 1999.
- [14] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.
- [15] T. Ly, D. Knapp, R. Miller, and D. MacMillen. Scheduling using behavioral templates. In *DAC*, San Francisco, 1995.
- [16] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *ICPP*, St. Charles, IL, 1993.
- [17] P. R. Panda, N. D. Dutt, and A. Nicolau. Exploiting off-chip memory access modes in high-level synthesis. In *IEEE Transactions on CAD*, Feb. 1998.
- [18] B. Prince. *High Performance Memories, New Architecture DRAMs and SRAMs evolution and function*. Wiley, West Sussex, 1996.

- [19] S. Przybylski. Sorting out the new DRAMs. In *Hot Chips Tutorial*, Stanford, CA, 1997.
- [20] Texas Instruments. *TMS320C6201 CPU and Instruction Set Reference Guide*.
- [21] Trimaran Release: <http://www.trimaran.org>. *The MDES User Manual*, 1997.
- [22] S. Wuytack, F. Catthoor, G. de Jong, B. Lin, and H. D. Man. Flow graph balancing for minimizing the required memory bandwidth. In *ISSS*, La Jolla, CA, 1996.