Differential Multithreading: Recapturing Pipeline Stall Cycles and Enhancing Throughput in Small-Scale Embedded Microprocessors

John Haskins, Jr. Kevin Skadron

Department of Computer Science
University of Virginia
Charlottesville, VA 22904

{predator|skadron}@virginia.edu

Abstract

This paper presents Differential Multithreading (dMT) as an inexpensive way to achieve high throughput from a single-issue architecture. dMT switches among multiple instruction streams in response to pipeline stall conditions but saves in-flight instructions, thus squashing pipeline bubbles and ensuring maximal utilization of a single pipeline. dMT uses auxiliary pipeline registers to save the state of in-flight but stalled instructions. This squashes bubbles that would otherwise arise from data hazards, branch delays, and cache misses.

This paper describes the pipeline organization necessary to support dMT, explains the advantage of shared-pipeline multithreading, and presents preliminary results which suggest that dMT can substantially increase processor utilization.

1 Introduction

Microprocessors for embedded devices must typically use simple organizations, precluding the use of many techniques for exploiting instruction-level parallelism (ILP) like multiple issue, out-of-order execution, and sophisticated branch prediction. Events like cache misses and read-after-write (RAW) data hazards therefore produce pipeline bubbles that waste time and power.

Previous research has shown multiple hardware contexts valuable for improving throughput, hiding memory latency and supporting both thread-level and instruction-level parallelism in CPU-intensive computations. Unfortunately, previous techniques for implementing multiple hardware contexts employ relatively heavy-weight mechanisms which preclude the deployment of these solutions into embedded devices. Indeed, many embedded processors still use a simple, single-issue architecture. This paper proposes *differential multithreading* (dMT), which multi-threads a single pipeline at an instruction-level granularity, and offers the potential to achieve many of the benefits of multithreaded architectures with only a modest increase in hardware complexity. In this paper we focus on how dMT will be applied to a single pipeline, but the technique is straightforward to extend to in-order multi-issue or VLIW systems.

The idea behind dMT is to squash pipeline stalls from one thread by executing another thread, thus increasing processor utilization. This is best illustrated by analogy to multitasking operating systems. Imagine a standard uniprocessor system, executing—for the sake of simplicity—just two tasks. So long as the first task, t_0 is executing normal instructions, it is allowed to run alone on the CPU, but when it executes a system call for some complex operation such as a disk read, the kernel will snapshot the system, preserving the context of t_0. Then, the kernel will restore the (previously snapshotted) second task, t_1, allowing it to execute while t_0's disk access proceeds in parallel. The insight here is that the kernel schedules the shared resource—the CPU—to maximize its utilization.

In a dMT architecture, the shared resource is the single execution pipeline. Imagine that this dMT architecture contains two independent instruction streams analogous to the two tasks of the aforementioned UNIX-like OS: say, i_0 and i_1. When the instructions fetched from i_0 enter a stall condition (*e.g.*, a RAW

Published in the Proceedings of the Workshop on Complexity-Effective Design, June 10, 2000, Vancouver, B.C.—held in conjunction with the 27th International Symposium on Computer Architecture.

data hazard), the dMT pipeline will immediately begin executing instructions from i_1, squashing the pipeline bubble that would otherwise result. In-flight instructions from i_0—except the one causing the stall condition and its successors—will continue to flow through the pipeline, committing their results to i_0's register file at the final stage. In-flight instructions from i_0 that must wait on the stall condition are snapshotted in auxiliary pipeline registers. Similarly, when instructions from i_1 enter a stall condition, the dMT pipeline will resume fetching and issuing instructions from i_0, starting with the instruction that caused the context switch to i_1. The very lightweight alternation among contexts in dMT is the reason for our naming it "differential" multithreading.

In a single-issue pipeline, the *maximum utilization* is 100%, meaning that 100% of CPU cycles are spent on useful computation with no stalls. Realistic implementations fall short of the maximum because of stalls in the pipeline which prevent the retirement of an instruction in every cycle. The objective of the dMT design is to asymptotically approach the maximum throughput of one instruction/cycle. dMT accomplishes this by increasing pipeline utilization, switching among multiple instruction streams in response to stall conditions. This is in contrast to other techniques (*i.e.*, larger, more highly associative caches, data forwarding) which increase pipeline utilization for only a single instruction stream.

Specific examples of embedded applications that can benefit from heightened throughput include: video game units requiring artificial intelligence manipulation of multiple computer-controlled characters, music playback, player input processing, data (pre)fetch from video game ROM, and multitasking portable personal organizers. The aforementioned applications are fairly sophisticated, but nonetheless are representative of embedded systems. In the case of the video game system, the largest gain comes from the fact that many threads of control are *independent* of one another and do not require synchronization. In the case of the multitasking personal organizer, independence does not necessarily come from multiple independent threads inside a single application, but from executing independent applications. Note that we use the terms "context" and "thread" interchangeably in this paper.

This paper makes a case for differential multithreaded architectures, describes how dMT manages the pipeline in order to squash stalls, and presents measurements showing the number of pipeline stall cycles that can be recaptured by dMT. The next section describes related work. Then Section 3 describes the organization of a dMT processor, Section 4 presents measurements showing the opportunity for dMT to increase pipeline utilization, and Section 5 concludes the paper.

2 Related Work

Traditional multithreaded (MT) architectures achieve performance gains by rapidly switching hardware contexts after each instruction. The Tera multithreaded architecture [2], for instance, has 128 hardware contexts and rotates execution among these contexts. Rotating execution in this fashion allows the Tera to hide latencies experienced by individual threads. Indeed, the time required to service all 128 contexts is greater than the time required to service memory requests, thereby masking memory access latency, and the Tera is thus completely cacheless.

Simultaneous multithreaded (SMT) architectures [7, 15] take a different approach, extending wide-issue superscalar architectures by allowing multiple hardware contexts to issue instructions to the execution units—in any given cycle, several instructions from a single thread might issue, or a mix of instructions from several different threads might issue. Subordinate simultaneous microthreading (SSMT) is an adaptation of simultaneous multithreading wherein *microthreads* are spawned—either explicitly or in response to processor events—to perform computation-specific optimizations on behalf of a single primary thread [4].

The closest technique to dMT of which we are aware is the APRIL system [1], which also context switches on a cache miss. Unfortunately, APRIL does not perform differential context-switching, and because it has a much more heavyweight context switch (10 cycles—the pipeline must drain) than dMT, it is not suitable for hiding primary-cache misses or other pipeline hazards. In a similar vein, Mowry and Ramkissoon [8] use informing memory operations to trap on cache misses and switch threads. Again, even though the informing memory operations make this context switch very lightweight, this technique does not save pipeline state and cannot achieve the same pipeline utilization as a dMT processor.

These technologies are not readily deployable into embedded systems that need the best possible performance and yet are still chiefly constrained by cost, size, and power. Microprocessors for such environments cannot yet afford the aforementioned types of multi-threading: they require too many hardware contexts, too much issue capability, and too many functional units.

On the other hand, even low-cost microprocessors can benefit from faster execution when it can be achieved affordably. Low-cost microprocessors are typically in-order issue and often only single-issue, yet this simplicity does not preclude their suffering from pipeline stalls due to data hazards and cache misses. (In fact, some systems contain no cache, in which case each memory reference introduces a stall. Examples include the Motorola DragonBall [9], used in Palm Pilots, and the Zilog Z80 [10], used in Nintendo GameBoys.) In addition, this simplicity means techniques like out-of-order issue, branch prediction, and data forwarding are not available to eliminate these stall cycles. dMT uses a low-cost version of multithreading to squash these pipeline bubbles, thereby achieving better utilization and improving system performance.

3 Design Overview

This section provides an overview of how dMT's functionality is implemented in the pipeline.

As a baseline, we use a classic five-stage, single-issue, RISC pipeline similar to the MIPS R2000. Note that we assume the presence of pipeline interlock hardware to detect data hazards and cache misses and stall the instruction stream accordingly. This differs from the R2000, which had no interlocking mechanism.[1] Like the R2000, our model processor has no forwarding network. Data forwarding can be expensive to implement, especially in constrained environments like embedded microprocessors. The hardware required to implement dMT will offer no savings compared to a forwarding network, but data forwarding can only reduce a single source of wasted cycles: read-after-write dependencies. In contrast, dMT recaptures many sources of pipeline bubbles.

We define a stall condition as the act of halting the pipeline in response to read-after-write data dependencies, control hazards and cache misses. With no data forwarding, the five-stage pipeline of the reference architecture exhibits two versions of the read-after-write data hazard; the first causes a one-cycle stall; the second causes a two-cycle stall.

The second major stall condition is that caused by a cache miss. When data is not available from the cache, the processor must wait until that data arrives before it can complete execution of the memory access instruction. If the processor contains no cache, then every memory reference introduces stall cycles.

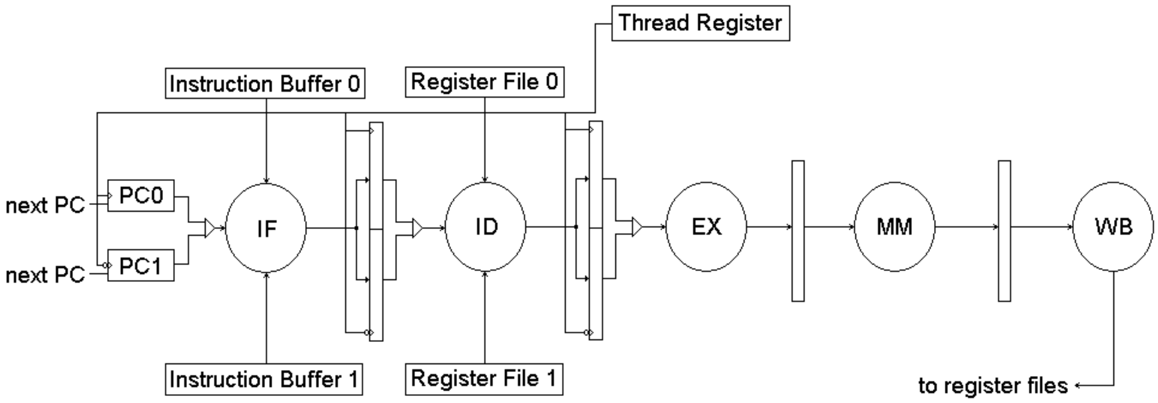

Figure 1: The dMT pipeline.

3.1 dMT Organization

We assume a dMT processor allows two threads to be active on the CPU at any given time. To implement dMT, the PC and the IF-ID and ID-EX pipeline registers must be duplicated, and the fetch stage must

[1] The name MIPS comes from the phrase: *Microprocessor without Interlocked Pipeline Stages* [5].

be augmented with at least a one-instruction fetch buffer. Each resulting "half" of the duplicated registers has a write enable whose setting is determined by which thread is currently allowed to make progress—see Figure 1. Each signal into these duplicated pipeline registers has a fanout of two–one copy into each half of the pipeline register. Only the half owned by the active thread is write-enabled; the other half holds the context of the stalled thread. Each signal into the next stage requires a multiplexor (mux), to choose the the correct pipeline input from one or the other half of the pipeline register. The register file must also be duplicated—one register file for each thread. The input muxes to the ID and EX pipeline stages and the extra register file will inevitably place pressure on the processor cycle time. Whether the consequent delay is affordable is one of the major open questions in this proposed scheme.

Stall detection is performed in the ID stage by the same interlock hardware used to implement interlocking on data hazards and cache misses. A thread that encounters a data hazard (*i.e.*, a RAW hazard) can be thought of as having two parts: a *committable* part—those instructions in the EX, MM and WB stages—and a *dependent* part—those instructions in the IF and ID stages. When interlocking hardware detects a data hazard, instructions in the committable part will be allowed to continue to flow through the pipeline unhindered; instructions in the dependent part however, will be frozen in their respective half of the pipeline registers. These frozen instructions will be able to continue execution once the final committable instruction—*i.e.*, the producer—commits in the WB stage. When control of the pipeline is returned to the previously-frozen instructions, the first dependent instruction—the one in the ID stage when the data hazard was detected—must be reissued into the ID stage. This is because ID fetches actual source values from the register file; since the data hazard prevented these source operands from being fetched before, they must be fetched now that the dependence has been resolved.

As with data hazards, threads which miss in the cache can be thought of as having committable and dependent parts. With cache misses however, the committable part consists of the instructions in the MM and WB stages of the pipeline, while the dependent part consists of those instructions in the IF, ID and EX stages. No more instructions from this thread may make forward progress until the memory reference is completed, so all dependent instructions are frozen in their respective half of the pipeline registers. Unlike data hazards, frozen dependent instructions after a cache miss are able to continue their execution after the memory reference is serviced and finished in the MM stage.

If dMT also treats control-flow hazards, it will require some form of pre-decode to detect control hazards (*i.e.*, conditional or indirect branches). The committable part after a control hazard consists of every instruction before and including the offending branch. The dependent part will remain "empty" because the IF stage will not fetch further instructions until the branch completes the EX stage.

When an executing thread encounters a stall condition and the pipeline registers snapshots its state, the other active thread is given an opportunity to proceed. If they are not also stalled, instructions from this second previously-frozen thread resume execution, allowing the second thread to make progress and fill in the stall cycles from the first thread.

When the second thread eventually stalls, its state is frozen once again and the first thread is given the opportunity to continue. Note that if a thread's dependences are unresolved, it cannot take over control of the pipeline because its instructions cannot make forward progress. In other words, if thread α stalls and relinquishes the pipeline, instructions from thread β will not be allowed to take over execution unless its currently-frozen instructions are ready to proceed. In our current model of just two active threads, if thread β is unable to make forward progress, dMT simply stalls.

3.2 Data Routing and Forward Progress

In systems that use a data cache, memory references in dMT will be serviced by a variety of a non-blocking cache: although a cache miss stalls the missing thread, the cache can still service accesses by other threads. The cache can easily be adapted to support true non-blocking accesses (*i.e.* hit-under-miss) or full blocking behavior. In the latter case, no thread can issue memory references when a cache miss is pending.

Note also that the data cache must either use physical addressing or a one-bit tag by which memory references may tag themselves as belonging to one context or the other. Similarly, all instructions must be tagged so that they will modify only one of the two general-purpose register files.

dMT's dual-threaded nature also demands that we modify our notion of an operating system context

cycle	IF	ID	EX	MM	WB	IF	ID	EX	MM	WB
1	$\frac{\alpha_4}{(\beta_1)}$	$\frac{\alpha_3}{(\beta_0)}$	$\underline{\alpha_2}$	$\underline{\alpha_1}$	α_0	α_4	α_3	α_2	α_1	α_0
2	$\frac{(\alpha_5)}{(\beta_1)}$	$\frac{(\boldsymbol{\alpha_4})}{(\beta_0)}$	$\underline{\boldsymbol{\alpha_3}}$	$\underline{\alpha_2}$	α_1	(α_5)	$(\boldsymbol{\alpha_4})$	$\boldsymbol{\alpha_3}$	α_2	α_1
3	$\frac{\beta_1}{(\alpha_5)}$	$\frac{\beta_0}{(\alpha_4)}$		$\underline{\alpha_3}$	α_2	(α_5)	(α_4)		α_3	α_2
4	$\frac{\beta_2}{(\alpha_5)}$	$\frac{\beta_1}{(\alpha_4)}$	$\underline{\beta_0}$		α_3	(α_5)	(α_4)			α_3
5	$\frac{\beta_3}{(\alpha_5)}$	$\frac{\beta_2}{(\alpha_4)}$	$\underline{\beta_1}$	$\underline{\beta_0}$		α_5	α_4			
6	$\frac{(\beta_4)}{(\alpha_5)}$	$\frac{(\boldsymbol{\beta_3})}{(\alpha_4)}$	$\underline{\boldsymbol{\beta_2}}$	$\underline{\beta_1}$	β_0	α_6	α_5	α_4		
7	$\frac{\alpha_5}{(\beta_4)}$	$\frac{\alpha_4}{(\beta_3)}$		$\underline{\beta_2}$	β_1	α_7	α_6	α_5	α_4	
8	$\frac{\alpha_6}{(\beta_4)}$	$\frac{\alpha_5}{(\beta_3)}$	$\underline{\alpha_4}$		β_2	α_8	α_7	α_6	α_5	α_4
9	$\frac{\alpha_7}{(\beta_4)}$	$\frac{\alpha_6}{(\beta_3)}$	$\underline{\alpha_5}$	$\underline{\alpha_4}$		α_9	α_8	α_7	α_6	α_5
10	$\frac{\alpha_8}{(\beta_4)}$	$\frac{\alpha_7}{(\beta_3)}$	$\underline{\alpha_6}$	$\underline{\alpha_5}$	α_4	α_{10}	α_9	α_8	α_7	α_6

a b

Figure 2: Dual-threaded dMT execution (a) versus single-threaded execution with no result forwarding (b). When this figure begins, the β instruction stream is stalled due to a data hazard while instruction stream α is making forward progress. At the end of cycle 2, a data hazard is detected between instructions α_3 and α_4; instructions α_{1-3} are allowed to proceed while $\alpha_{4,5}$ are frozen. In cycle 3, instruction stream β is allowed to proceed starting with β_0 being redecoded in ID. When α_3 commits at the end of cycle 4, α_4's dependence is resolved; thus, in cycle 6 when a data hazard is detected between β_2 and β_3, thread α is ready to reassume control of the pipeline in cycle 7 when α_4 is redecoded.

switch. One mechanism to guarantee forward progress is to have the operating system swap out the older of the two hardware contexts, replacing it with the highest-priority thread from the ready queue.

3.3 Example

Figure 2a presents an example of two threads' executions, interleaved to hide each others' stalls, and Figure 2b presents an example of a single thread on a non-dMT CPU without result forwarding. In both figures, the pipeline stages are depicted horizontally and time is depicted on the vertical axis. In Figure 2a, we depict dMT's dual-threaded execution as follows: the numerators contain instructions currently active in the pipeline, while the denominators depict instructions that are currently frozen. The WB stage only contains a single instruction, because instructions which reach this stage will complete.

From the examples in Figures 2a and 2b, we can see the benefits of dMT. Stall cycles that would result in conventional single-issue execution are eliminated by our technique of "differentially" multithreading the single pipeline. Of course, Figures 2a and 2b assume the absence of a result-forwarding network. Yet even in a processor that implements forwarding, dMT can still play a substantial role by hiding cache misses, and especially in low-cost systems that omit a cache.

We also wish to point out that Figure 2 actually presents dMT in an unnecessarily unfavorable light. In the interest of clearly demonstrating *how* dMT works, we have made the pipeline stalls occur so often that dMT only improves throughput by one instruction (which is actually an attractive 10% improvement!). We expect that in typical execution sequences, dMT can produce further gains. This will be especially true for longer cache-miss times and for longer pipelines, where it becomes attractive to hide branch delays.

Figures 2a and 2b also suggest that dMT enhances utilization at the expense of single-thread performance—α makes more progress in Figure 2b than in Figure 2a. Yet if we carried this example farther, we would see that thread α regains lost ground as thread β stalls. In any case, processes' rate of forward progress often matters as much as single-thread performance. If single-thread performance is sufficiently important, the thread-switching policy can be changed to re-awake a stalled thread as soon as its stall condition resolves. In such a scenario, one thread always has priority, and the second thread is an auxiliary thread used only to recapture stall cycles. Priority shifts gradually among threads according to the operating system's scheduling policy.

Note also that processors with longer pipelines will require additional "duplicated" pipeline registers. This increases the total extra cost of dMT, since more pipeline registers must be duplicated along with the associated muxes. But as pipelines grow longer, branch costs grow as well, necessitating branch prediction unless dMT can hide these costs.

3.4 dMT and Branch Prediction

This paper does not treat the role of branch delays, because we assume that a classic, 5-stage RISC pipeline will use delayed branches and earliest-possible branch resolution to avoid the need for branch prediction while simultaneously hiding branch delays.

But with longer pipelines, as in the 14-stage UltraSPARC-III [6, 13], delayed branches no longer suffice. The conventional solution to hiding branch-resolution delays is to implement branch prediction. Static prediction [12] only eliminates 60-70% of branch delays [16], and most dynamic schemes require expensive hardware. Even a simple bimodal predictor [12] is typically the size of a small cache.

dMT provides an alternative. Instead of an expensive hardware-based branch predictor, the hardware can detect conditional branches and indirect jumps in the earliest possible stage (or use I-cache pre-decode information) to freeze the thread containing a branch and switch to the other active thread. The branch will continue through the pipeline and resolve, while giving the other thread's instructions an opportunity to make forward progress.

In longer pipelines, branch delays therefore provide a further major opportunity for dMT to hide pipeline stalls.

4 Pipeline Stall Statistics

4.1 Methodology

These data were obtained using release 3.0a of the University of Wisconsin's *SimpleScalar* software package [3]—the *sim-cache* program, in particular—running five SPECint95 [14] benchmark programs: *gcc*, *go*, *ijpeg*, *li* and *perl*. The benchmarks were compiled with gcc 2.6.3 and -O3 optimization for the SimpleScalar PISA instruction set. An important caveat is that the compiler was unaware of the lack of a forwarding network and therefore made no attempt to schedule instructions accordingly.

It should also be emphasized that the programs used to test the dMT concept were *not* setup to model dMT's multiple contexts. Furthermore, the SPEC benchmark programs used in this study are not necessarily representative of the software which would likely run on in the sort of embedded system for which a single-issue dMT processor would be suitable. The SPEC programs should merely be regarded as proxies for more appropriate benchmarks, used to permit us to get initial estimates of data-hazard and cache miss frequencies.

The reference architecture being modeled was an R2000-like, 5-stage pipeline. All simulations were fast-forwarded according to the methodology in [11] in order to avoid unrepresentative initial behavior, and statistics were gathered for the next 100 million instructions. Benchmark details appear in Table 1.

	Warmup insts	Input	Conditional branch static sites	dynamic refs
go	925 M	9stone21	4,627	11.2 M
gcc (cc1)	220 M	cccp.i	14,245	14.7 M
li (xlisp)	270 M	9queens	271	15.4 M
ijpeg	823 M	vigo.ppm	657	5.1 M
perl	600 M	scrabble	352	12.9 M

Table 1: Benchmark summary.
"Warmup insts" indicates the length of the preliminary phase of simulation, before statistics-gathering.

We assume that in the absence of other stalls, all instructions take one cycle to execute. Cache misses have a five-cycle latency, and for measurements in a cacheless system, all memory references are optimistically assumed to take five cycles. Many systems will in fact suffer much longer delays. The data cache in our experiments is always a 32 KB, 4-way set associative cache with 32-byte lines.

Our measurements characterize the contribution of (1) cache misses, (2) RAW data hazards that cause a one-cycle stall, and (3) RAW data hazards that cause a two-cycle stall to the overall execution time. These stall cycles represent opportunities for dMT to recapture wasted cycles and improve utilization.

Because even a cacheless system offers so many design possibilities for the fetch engine, in this study we focus on data-dependence effects; we therefore assume ideal instruction fetch and present no statistics regarding instruction-cache misses.

4.2 Results

The graphs in Figure 3 show the opportunities for dMT. It is important to note that in each graph, "100%" represents the total execution time for that particular processor configuration.

Figure 3a shows the fraction of execution cycles spent stalled on RAW data hazards and cache misses in a processor without result forwarding; Figure 3b shows the the fraction of execution cycles spent stalled on cache misses in a processor with forwarding. Figure 3c is similar to Figure 3a: it shows the fraction of execution cycles spent stalled on RAW data hazards and memory references in a processor that lacks result forwarding, and this time also omits a cache. Finally, Figure 3d shows time spent stalled on memory references in a processor with a forwarding network but no cache.

Figure 3a shows that a processor with no data forwarding spends approximately 40–50% of its time stalled on RAW data hazards and cache misses; Figure 3c shows that a processor with no data forwarding and no cache spends approximately 70% of execution cycles stalled. If dMT can recapture these stall cycles, it can double or triple utilization.

On the other hand, in the system with data forwarding and a cache, almost all the stall cycles come from cache misses. These are few due to our choice of a large, 4-way associative cache. Figure 3b shows that a dMT processor with forwarding and with such an aggressive cache can at most improve utilization by a few percent.

Most important, however, is Figure 3d. It shows that in low-cost embedded processors without a cache, even when data forwarding is present, 55–70% of execution cycles are spent stalled on memory accesses. Again, if dMT can recapture these wasted cycles, it can double or triple utilization.

While most processors today include data forwarding, the requisite hardware cannot help with other sources of stalls. dMT can attack data-cache misses, data hazards, and also instruction-cache misses and branch delays. dMT therefore offers the potential to maximize pipeline utilization with a modicum of extra hardware, and simultaneously allows the designer to *remove* other hardware like the data cache and the data-forwarding network.

5 Conclusions and Future Work

In this paper we have presented the idea of differential multithreading (dMT), an inexpensive technique for sharing a single pipeline between multiple hardware contexts in a way that hides each others' stall cycles. Whenever a thread encounters a stall condition, its dependent part relinquishes the pipeline while instructions in its committable part are allowed to continue to move through the pipeline. To fill in the resulting stall cycles, a second thread may execute. Making the pipeline available to another context when a stall condition occurs increases opportunities for pipeline utilization and reduces wasted cycles. dMT's utility is predicated on the assumption of a simple, low-cost processor like those found in many embedded systems. Preliminary data suggest numerous opportunities in such CPUs for dMT to reduce stall cycles and increase utilization.

In this initial description of dMT and shared-pipeline multithreading, we have used SPECcpu95 benchmarks rather than a more realistic set of embedded-system benchmarks, and we have only counted opportunities for dMT to squash stall cycles. The next step in exploring dMT is to find a suitable set of benchmarks and determine how much dMT actually improves performance, utilization and throughput . dMT requires at

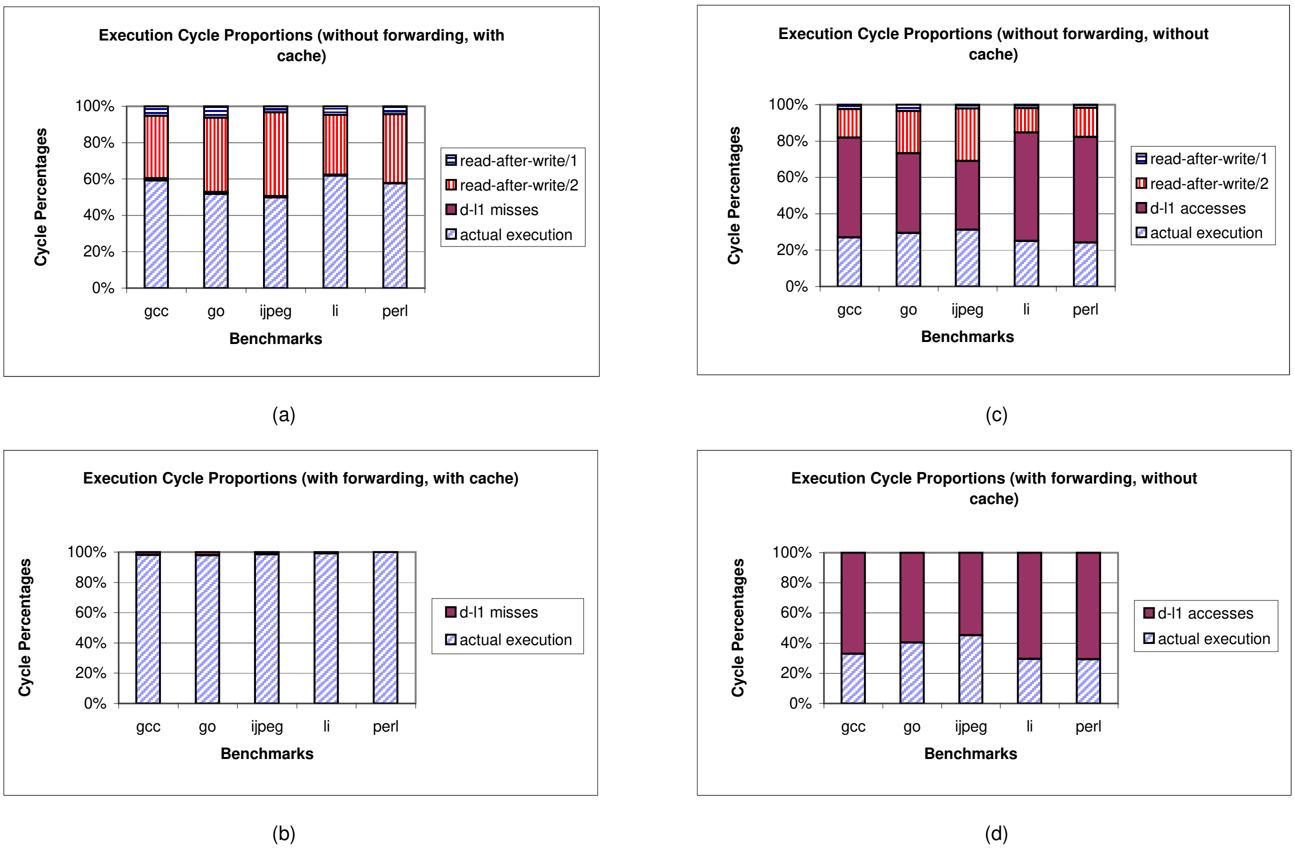

Figure 3: Stall-cycle statistics. (a) and (b) give results for a system with a 32 KB, 4-way cache, while (c) and (d) give results for a cacheless system. (a) and (c) use no data-forwarding network; (b) and (d) do use forwarding. It is important to note that in each graph, 100% represents the total execution time given that particular processor configuration. The total execution time is then broken down: "read-after-write/1" represents percentage of cycles lost to 1-cycle RAW-hazard stalls, "read-after-write/2" represents percentage of cycles lost to 2-cycle RAW-hazard stalls, "d-l1 misses" represents percentage of cycles lost due to misses in the level-1 data cache, and "d-l1 accesses" represents percentage of cycles lost due to memory accesses in a cacheless configuration.

least two simultaneously-runnable threads, so such an evaluation will require a system that accurately models system-level effects. We must also quantify the cost of supporting dMT, in order to determine whether the extra hardware required can be used more profitably elsewhere, and determine the extent to which the extra hardware for dMT places pressure on the cycle time. Further issues to explore include support for more than two simultaneously-active threads, support for multi-issue processors, and the best organization for the instruction and data caches. Finally, as stated before, deeper pipelines may present further opportunities to reduce wasted cycles; we will therefore explore pipelines having more than the five stages used here.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proc. of the 17th International Symposium on Computer Architecture*, May 1990.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. of the International Conference on Sumpercomputing*, May 1990.

[3] T. M. Austin. SimpleScalar release 3.0a. Computer software, http://www.cs.wisc.edu/mscalar/simplescalar.html, 1998.

[4] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proc. of the 26th International Symposium on Computer Architecture*, pages 186–95, May 1999.

[5] R. B. K. Dewar and M. Smosna. *Microprocessors: A Programmer's View.* McGraw-Hill, 1990.

[6] T. Horel and G. Lauterbach. UltraSPARC-III: Designing third-generation 64-bit performance. *IEEE Micro*, pages 73–84, May-June 1999.

[7] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen. ILP versus TLP on SMT. In *Proc. of Supercomputing 1999*, November 1999.

[8] Todd Mowry. Software-controlled multithreading using informing memory operations. Technical Report CMU-CS-98-169, School of Computer Science, Carnegie Mellon University; http://www.cs.cmu.edu/~tcm/tcm_papers/sw_mt_tr98.ps, October 1998.

[9] Motorola Semiconductor Products. Motorola's DragonBall(tm) integrated processor family. Web site, http://www.mot.com/SPS/WIRELESS/products/DragonBall.html, 1999.

[10] Thomas Scherrer. Thomas Scherrer Z80-family official support page. Web site, http://www.geocities.com/SiliconValley/Peaks/3938/z80_home.htm, 2000.

[11] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–81, November 1999.

[12] J. E. Smith. A study of branch prediction strategies. In *Proc. of the 8th International Symposium on Computer Architecture*, pages 135–48, May 1981.

[13] P. Song. UltraSparc-3 aims at MP servers. *Microprocessor Report*, pages 29–34, Oct. 27 1997.

[14] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. WWW site: http://www.specbench.org/osg/cpu95, December 1999.

[15] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of the 23rd International Symposium on Computer Architecture*, May 1996.

[16] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. of the 19th International Symposium on Computer Architecture*, pages 124–34, May 1992.