# dMT: Inexpensive Throughput Enhancement in Small-Scale Embedded Microprocessors with Differential Multithreading: Extended Results

UNIV. OF VIRGINIA DEPT. OF COMPUTER SCIENCE TECH. REPORT CS-2003-18

Kevin R. Hirst* John W. Haskins, Jr.† Kevin Skadron‡

Department of Computer Science
University of Virginia
151 Engineer's Way, Box 400740 Charlottesville, VA 22904-4740
USA
krh2n@alumni.rice.edu, predator@super.org, skadron@cs.virginia.edu

Oct. 2003

**Abstract**

*This paper examines differential multithreading (dMT) as an attractive organization for increasing throughput in simple, small-scale, pipelined processors like those used in embedded environments. dMT copes with pipeline stalls due to hazards and data- and instruction-cache misses by using duplicated pipeline registers to run instructions from an alternate thread. Results show that dMT boosts throughput substantially and can in fact* replace *dynamic branch prediction or can be used to reduce the sizes of the instruction and data caches. This report expands upon [1] by presenting extended results for additional dMT configurations.*

## 1 Introduction

Previous research has demonstrated the effectiveness of multiple hardware contexts for improving throughput, hiding memory latency and supporting thread-level and instruction-level parallelism in CPU-intensive computations. *Differential multithreading* (dMT) is a low-cost version of hardware multithreading in which multiple instruction streams share a single pipeline, and the processor squashes pipeline stalls from one thread by executing instructions from another thread. (In this paper, when we refer to a "thread-switch", we mean a pipeline switch between its active instruction streams and not an OS-level switch among kernel threads.) These threads might be separate, independent processes or cooperating threads within a single process. A simpler version of this technique—*block multithreading* or BMT—was first described by Farrens and Pleszkun in [2]. dMT extends BMT by adding the ability to switch active threads in response to misses in the instruction- and data-caches.

**Benefits.** Conventional pipelines fall short of maximum throughput because stalls in the pipeline prevent the retirement of an instruction in every cycle. The objective of differential multithreading, like block multithreading, is to asymptotically approach the maximum throughput of 1 IPC by switching among multiple instruction streams in response to stall conditions. This is in contrast to other techniques (*e.g.*, larger, more highly associative caches or data forwarding[1]) which increase pipeline utilization for only a single instruction stream.

Although single-issue ("scalar") organizations are no longer used in high-performance processors, they remain common even in new processor designs for small-scale, embedded devices. Some embedded processors in fact

*Now with Kidwell McGowan Associates, Sterling, Va.
†Now with the Institute for Defense Analyses Center for Computing Sciences, Bowie, Md.
‡Contact author
[1]Bypassing data from later instructions to earlier ones to avoid data hazards.

omit the data-cache altogether, a configuration for which differential multithreading is especially valuable. Overall, multithreading benefits embedded workloads for which throughput is as important as single-thread execution time, and in particular, workloads that would run on the simpler, single-issue base architecture considered in this paper and are prevalent in embedded environments. Specific examples include embedded applications like video game units, portable personal organizers, and process-control systems. Block multithreading and differential multithreading are especially useful for multithreaded workloads. Video game systems are an example: they must simultaneously support artificial intelligence manipulation of multiple computer-controlled characters, background music playback, player input processing, and data (pre)fetch from the game ROM. Unfortunately, finding a suitable, non-proprietary multithreaded workload has proven difficult, and our evaluations focus on throughput of multiple, semi-independent applications.

Not only does differential multithreading increase throughput, it does this so effectively that other processor structures, like caches or branch predictors, can be made smaller or even eliminated. For example, Section 5 shows that dMT allows the use of a smaller cache or the complete elimination of branch-prediction hardware. In [3], we also found that dMT is highly effective in chip-multiprocessor configurations, more so than non-multithreaded dual-issue processor cores.

**Contributions.** Our contribution stems from our extensions to block multithreading, which can be thought of as combining aspects of the BMT [2] and Runahead [4] pipelines. BMT uses dual-decoder logic to perform instruction interleaving at the end of decode, where the issue logic selects one instruction to promote to the next stage of the pipeline. Placing the interleaving mechanism solely in this early stage of the pipeline allows BMT to respond to data hazards and long-latency branch delays; it cannot, however, avoid stalls that result from misses in the instruction or data caches. Our implementation—dMT—gains this ability by capturing and storing in-flight instructions when an instruction stream encounters a stall for all sources of pipeline stalls.

A further contribution comes from re-examining block multithreading in light of more modern benchmarks. The original BMT work used only the Livermore Loops. Our studies use the MiBench [5], MediaBench [6], and SPECint95 [7] suites.

The remainder of the paper is organized as follows. The next section discusses related work, Section 3 gives details about the design of the dMT pipeline, Section 4 discusses our experimental methodology, Section 5 presents our results, and Section 6 concludes the paper.

## 2 Related Work

Our work is most closely related to BMT [2]. It describes three different policies for interleaving instructions: *every-cycle*, *blocked* and *prioritized*. Every-Cycle switches threads after every clock cycle, and Blocked only in response to stalls. For Prioritized, the thread with priority resumes execution as soon as its stall condition resolves, regardless of the status of the other thread; this policy is useful for real-time workloads. By resuming its execution as soon as its stall condition resolves, the thread with priority will incur no penalty beyond what it would if it were executed alone, thus preserving the predictability of its execution time. We find that the Blocked and Prioritized policies give fairly similar performance for both BMT and dMT, and that Every-Cycle is consistently the worst.

Traditional multithreaded architectures such as the Tera [8] achieve performance gains by every-cycle scheduling. Every-cycle scheduling among the Tera's 128 hardware contexts allows the Tera to hide latencies experienced by individual threads, and indeed, the time required to service all 128 contexts masks memory latency and permits the Tera to be completely cacheless. Simultaneous multithreaded (SMT) architectures [9, 10] 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, a mix of instructions from several different threads might issue. Neither the Tera nor SMT is readily deployable into an embedded system that needs high performance and yet is still constrained by cost, size and power. In particular, it is not obvious how to cost-effectively extrapolate the Tera and SMT approaches to a scalar, single-issue pipeline.

Two more systems that are related to block multithreading are APRIL [11] and Runahead [4]. Like dMT, APRIL thread-switches on a cache miss; however, APRIL uses a more heavyweight thread switch in which the pipeline must drain. The consequent 10-cycle delay is not suitable for hiding pipeline stalls. Runahead microprocessors speculatively execute instructions past a first-level D-cache miss. These instructions are not committed; their purpose is to uncover subsequent memory instructions whose target address is calculable. Even though the result of these references is discarded, they serve as lightweight prefetches. Unlike dMT, Runahead does not attempt to fill in the stall cycles with instructions from another instruction stream.

Low-cost microprocessors are typically in-order issue and often only single-issue, yet this simplicity does not preclude their suffering from pipeline stalls. In fact, some processors contain no cache, in which case each memory reference introduces a stall. Examples include the Motorola DragonBall [12] used in Palm Pilots and the Zilog Z80 [13] used in Nintendo Gameboys.

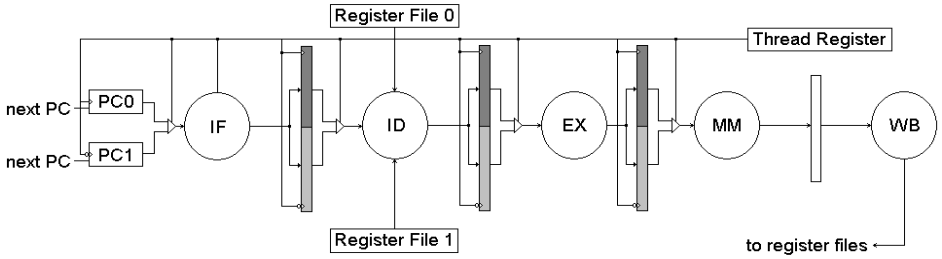


Figure 1: The dMT pipeline for a two-way dMT organization. (Reproduced from [3], ©2001, IEEE.) In reality, to minimize wiring cost, the pipeline register would not be partitioned as shown, but rather bit-sliced: each signal would have two possible sources that are stored adjacent in the pipeline register.

## 3 dMT Design Overview

This section provides a more detailed description of the dMT organization. The baseline architecture for our design uses the classic five-stage, single-issue RISC organization of the ARM10 [14].

For this discussion, we assume a dMT processor that allows two threads to be present on the CPU at any given time. The identity of the current active thread is held in the *thread register*. Simultaneously hosting two or more instruction streams requires that the program counter and register file be replicated—one per hardware context. Most importantly, (as discussed in Section 2), dMT duplicates the pipeline registers between the fetch-decode (IF-ID), decode-execute (ID-EX) and execute-memory (EX-MM) stages. These duplicated pipeline registers are used to capture in-flight instructions. If an instruction entering the WB stage will not stall, as we assume in this work, then the MM-WB pipeline register does not need to be duplicated (recall that in any particular stage, only one instruction is active per cycle). The dMT pipeline organization is depicted in Figure 1. Not shown is the fact that the instruction in the WB stage needs a separate tag to indicate which thread it belongs to and hence which register file to write to. This is necessary because the instruction in the WB stage may be from a different thread than previous stages.

Each data and control signal entering the duplicated pipeline registers has a fanout of two: one copy into each "half" of the pipeline register. This would probably be implemented in a bit-sliced fashion, with the two sources of each output signal co-located to minimize wiring length. This can be thought of as taking the two logical copies of the pipeline register and interleaving them. Each logical "half" of the duplicated registers has a write-enable whose setting is determined by the active thread identified in the thread register. Only the half owned by the active thread is write-enabled; the other half holds the state of the stalled thread. Each signal entering the next stage requires a multiplexor to choose the correct pipeline input from one or the other half of the pipeline register. The thread register is also used to control these multiplexors. Processors with longer pipelines will require additional duplicated pipeline registers. This increases the total cost of dMT, since more pipeline registers must be duplicated along with the associated multiplexors. But as pipelines grow longer, branch costs grow as well, necessitating branch prediction unless dMT is used to hide these costs.

In dMT, a thread that encounters a stall condition can be thought of as having two parts: a committable part and a dependent part. Instructions in the committable part are unaffected by the stall and continue to flow through the pipeline unhindered; instructions in the dependent part cannot proceed until the stall is resolved, and are captured and held in their respective half of the pipeline registers. These frozen instructions continue execution once the offending instruction (the producer) completes and the dependence is resolved. We assume that a stall condition can be detected sufficiently early so the new thread can take over execution immediately after the stall condition is detected,

with no wasted cycles. Although stall conditions are probably detected late in each stage, the only additional time required is the propagation delay to distribute the choice of thread for the next cycle in time to set up the multiplexors at the beginning of the next stage. If this requires an additional cycle, and introduces one stall cycle each time a thread is switched, the benefits we report decline significantly. Note that no additional logic is required to detect stall conditions: the same logic that detects hazards in conventional processors is leveraged by dMT. The only exception is the possibility of cross-thread accesses to the same location in memory, which requires extra logic to possibly stall the second thread—see below.

A thread switch affects only those stages in the dependent part; all these stages switch in unison immediately. A thread switch does not affect those stages in the committable part. For example, on a data-cache miss, all instructions after the cache are potentially data dependent. The dependent part therefore consists of instructions in the IF, ID and EX stages; the committable part consists of the pending memory operation and the instruction in the WB stage. When the data cache detects a miss, dependent instructions are frozen in their respective half of the pipeline registers. As another example, a branch creates a control hazard that prevents further fetching from that thread. An I-cache miss similarly prevents further fetching. In these cases, the committable part consists of all instructions currently in the pipeline (including the branch in the case of the control hazard). The dependent part simply consists of the PC of the stalled thread—the PC can be thought of as another pipeline register preceding the fetch stage. If no thread's stall condition is resolved, the pipeline experiences a true stall until one thread or the other can proceed. A detailed itemization of possible stalls and their treatment can be found in [3].

Since some exceptions and many external interrupts require all active threads to be suspended, it is probably easiest to always suspend all threads on any kind of exception or interrupt. This means that both register files and PCs must be saved, which will modestly increase the cost of handling an exception. Since exceptions should be rare, the extra tens of cycles per exception should be negligible.

The input multiplexors to the ID, EX and MM pipeline stages and the extra register file will inevitably place pressure on the processor cycle time. The multiplexor can actually be implemented as a wired-OR of the tri-state-enabled possible outputs, so one extra gate delay is a reasonable expectation for the associated overhead. A useful rule of thumb is that the cycle time might be lengthened by as much as ten percent for every extra gate delay [15]. The detailed design of these multiplexors and their associated overhead is subject to a variety of design choices. A detailed implementation is beyond the scope of this paper, so we focus on cycle-level simulations. Some of the increased throughput from dMT's ability to recapturing stall cycles will be offset by the increase in cycle time. For example, if we indeed assume that dMT's clock runs ten percent slower, then throughput improvements of about 10% or better will likely exhibit actual speedups. In other words, the increase in throughput can be thought of as the break-even point in terms of how much reduction in clock speed (due to the extra dMT hardware) can be tolerated by dMT before it performs worse than a conventional organization.

In this paper, we assume that dMT configurations never include branch prediction. Our focus is on throughput and we found that the branch predictor conferred minimal benefit from this standpoint. Indeed, as we show later, a dMT configuration without branch prediction does better than a non-dMT configuration with branch prediction! Branch prediction does impact end-to-end execution time of a single thread, but misprediction handling in dMT configuration is more complex than in a conventional pipeline: either rolling back all extant threads (contravening our throughput goal), or requiring extra hardware to squash only mis-speculated instructions in the mispredicting thread. Adding and evaluating branch prediction in a dMT pipeline is an interesting area for future work.

Finally, it is worthwhile to comment on the impact of dMT on multi-threaded programs. From a correctness standpoint, a dMT implementation is no different than any other MT or CMP processor; namely consistency must be enforced by appropriate synchronization. Coherence, on the other hand, is a non-issue here, because the dMT threads share a common cache. Accesses to the same physical address (for example, the first thread may take a cache miss when reading some location, and while that thread is stalled, the other thread may attempt to write to the same location) do need to be identified but can be treated as data hazards. From a performance standpoint, it is true that synchronization delays will sacrifice some of the reported improvements in throughput. But even for many MT programs, these delays should not be ubiquitous to the point where all the improvement is forfeited. This paper focuses on the performance of independent programs on a dMT machine; evaluating multi-threaded programs is an interesting area for future work.

## 4 Simulator and Benchmarks

We model BMT and dMT by using Wisconsin's *SimpleScalar* 3.0a software package [16]. We assume that in the absence of other stalls, all instructions take one cycle to execute. Our baseline assumption is that the cache miss penalty for the first-level cache is five cycles (to a second-level cache or to some form of embedded DRAM); later we also consider a longer miss penalty of ten cycles.

We use a mix of benchmark programs from the SPEC95 [7], MediaBench [6], and MiBench [5] suites, and we also used the Dhrystone benchmark [17]. Rather than show data averaged across all benchmarks as in [2], we chose pairs of benchmarks to run together. This lets us show a richer variety of reactions to BMT and dMT; some benchmarks have very distinctive behaviors. Naturally, it was impossible to present data for all possible pairs, so we chose a subset of the benchmarks from each suite and selected the most sensible pairs that we could derive—like a game and an image utility (*go* and *ijpeg*) or a speech compression/decompression tool and an image compression/decompression tool (*gsmencode* and *epicencode* / *gsmdecode* and *epicdecode*). Despite its well-known drawbacks, we also include the Dhrystone benchmark because it has been used so much for benchmarking embedded microprocessors. We chose to use SPEC95 over SPEC2000 [18] because we were specifically interested in *go* (a game and also a program known for its poor branch behavior) and *ijpeg* (an image-processing program).

Except for *epic, jpeg, pegwit, dhrystone* and *MiBench*, which are short and were run to completion, all simulations were fast-forwarded according to the methodology in [19] (for SPEC) or 100 million instructions (for longer MediaBench programs) in order to avoid unrepresentative initial behavior, and statistics were gathered for the next 100 million instructions. Because all programs run for approximately the same length in our simulations, we report results using simple arithmetic means.

The benchmarks were compiled with gcc 2.6.3 and -O3 optimization for the SimpleScalar PISA instruction set.

It would also be interesting to evaluate the benefits of dMT from an energy-efficiency standpoint. Although the extra register file and pipeline register bits will increase power dissipation, higher throughput will reduce total execution time and hence total energy consumed for a given workload. A detailed energy evaluation is another interesting area for future work.

## 5 Results

### 5.1 Experimental Configurations

To identify the different configurations we explore, each is named in the pattern XXX-YYY, where XXX is either "base" or "MT" and YYY indicates the processor organization. The same configuration is used for both BMT and dMT. The following abbreviations are used:

| | |
|---|---|
| 0: | I-cache only |
| f: | forwarding |
| C: | large cache configuration<br>(16-KB I-cache and 8-KB D-cache, both four-way) |
| c: | small cache configuration<br>(8-KB I-cache and 2-KB D-cache, both two-way) |
| b: | dynamic branch predictor |

So *base-Cfb* is an ARM10-like, 5-stage, single-issue processor with forwarding, a 16-KB I-cache and 8-KB D-cache, and a bimodal branch predictor; *MT-cf* is a multithreaded processor (either BMT or dMT) with forwarding, an 8-KB I-cache and 2-KB D-cache, and no branch prediction. When a dynamic predictor is used, it is a 2-bit bimodal predictor with a 128-entry branch target buffer, as in the ColdFire v4 [20].

We present data for a total of 14 different pairings of 18 different benchmark programs. We used these pairs to make ten different comparisons of non-multithreaded and multithreaded processors. For each we present a graph (Figures 2–5) that compares their IPCs for each of the benchmark pairs. In each comparison, various multithreaded organizations are compared to a comparably configured or superior baseline configuration. Figure 2 compares three systems with equivalent cache, branch prediction, and forwarding (f vs. f, etc.). These show how much extra throughput is obtained by simply adding dMT to an existing design. The benefits are especially notable for a configuration without data caches. Figure 3 adds the ColdFire's dynamic branch predictor to each baseline configuration, but for each of these comparisons, MT still omits branch prediction, switching threads instead. These results demonstrate MT's robustness against control hazards. Figure 4 presents one comparison where the baseline system uses the larger cache sizes listed above, while the MT systems use the smaller. This demonstrates dMT's capability to reduce the need for larger

caches. Note that some configurations appear in more than one comparison; this is in order to illustrate the tradeoffs multithreading permits. In addition, we present data for some of the same pairings but varying cache miss penalty in Figure 6, and present data for six quadruplets of SPEC95 and MediaBench applications in Figure 7 to show the performance of four-way dMT. Table 1 gives the average improvement in throughput (IPC) for each comparison.

It is useful to compare the hardware overhead of dMT, although this is difficult to do since the area overhead is heavily dependent on myriad design choices. But we can count the number of bits of each type—pipeline register, register file, and cache. We estimate that each set of copies of the pipeline registers requires 1,237 bits, so the non-dMT implementation uses 1,237 bits, the two-way dMT uses 2,474, and the four-way uses 4,948. For a 32-bit machine, we estimate that each copy of the register file costs 2048 bits. Among hardware that is not duplicated, the branch predictor requires 8192 bits; the "c" cache configuration requires 91,136 bits, and the "C" configuration requires 215,808 bits. These estimates assume MESI state bits in the cache and no ECC bits in the cache or register file.

| Baseline Config. | MT Config. | Graph | Δ IPC | Δ IPC(dmt10) | Δ IPC(4X) |
|---|---|---|---|---|---|
| base-f | MT-f | Fig. 2 | 73.2% | 80.2% | 139.0% |
| base-cf | MT-cf | Fig. 2 | 24.5% | 9.4% | 43.9% |
| base-Cf | MT-Cf | Fig. 2 | 22.6% | 15.2% | 33.8% |
| base-cfb | MT-cf | Fig. 3 | 4.2% | -6.0% | 23.6% |
| base-Cfb | MT-Cf | Fig. 3 | 2.6% | -4.0% | 12.9% |
| base-cf | MT-f | Fig. 4 | -4.9% | -23.7% | 37.8% |
| base-Cf | MT-cf | Fig. 4 | 10.4% | -6.4% | 17.2% |
| base-Cfb | MT-cf | Fig. 4 | -7.6% | -22.4% | 8.7% |
| base-f | MT-0 | Fig. 5 | 44.3% | 48.7% | 84.7% |
| base-cf | MT-0 | Fig. 5 | -20.7% | -37.2% | 12.3% |

Table 1: Baseline vs. MT comparisons. ΔIPC columns give the throughput improvement for 2-way dMT with a 5-cycle miss penalty, 2-way dMT with a 10-cycle miss penalty, and 4-way dMT with a 5-cycle miss penalty.

The MT pipelines run both instruction streams simultaneously. In contrast, the baseline architecture runs the two benchmarks back-to-back. To obtain the most comparable results across organizations, we terminate the MT simulations when one instruction stream terminates or when it completes its simulation allotment of 100 million instructions. The baseline simulation then runs its two programs for exactly the same number of instructions as the MT simulation.

We also measured the frequency with which dMT (Blocking policy) switches threads. Arithmetic means taken across our set of benchmark pairs are reported in Table 2. As expected, with no forwarding (the "0" configuration), thread switches are frequent (every 1.7 instructions), and as the configuration becomes more aggressive, thread switches become less frequent (reaching a level of every 5.3 instructions with "Cf").

| dMT configuration | insts/switch |
|---|---|
| 0 | 1.72 |
| f | 2.22 |
| cf | 3.72 |
| Cf | 5.28 |

Table 2: Instructions between dMT instruction-stream switches (arithmetic mean).

### 5.2 Discussion

For equivalent cache, branch-prediction, and forwarding configurations (Figure 2), both dMT choices obtain dramatically better throughput than the non-multithreaded organization. This is perhaps not a fair comparison, because dMT has a small amount of extra hardware. But it shows that dMT does indeed recapture stall cycles and uses them to boost throughput, and it shows that adding dMT can improve throughput substantially—dramatically so for a design without data caches. For two-way dMT, it boosts throughput by 73% in a processor with no data cache, by more than 24% in the small-cache configuration, and by slightly less than 23% in the large-cache configuration, all of which are certainly more than any performance loss due to extra latency introduced by the multiplexors after pipeline stages. For dMT, sometimes the Blocking policy outperforms the Prioritized policy and vice-versa, but overall, the difference is small, a few percent. This is due to specific reactions to cache contention. The prioritized policy will be useful for real-time workloads where a specific thread must complete in a specified amount of time or requires some other determinism. The prioritized policy replicates dedicated pipeline behavior for the prioritized thread except for the

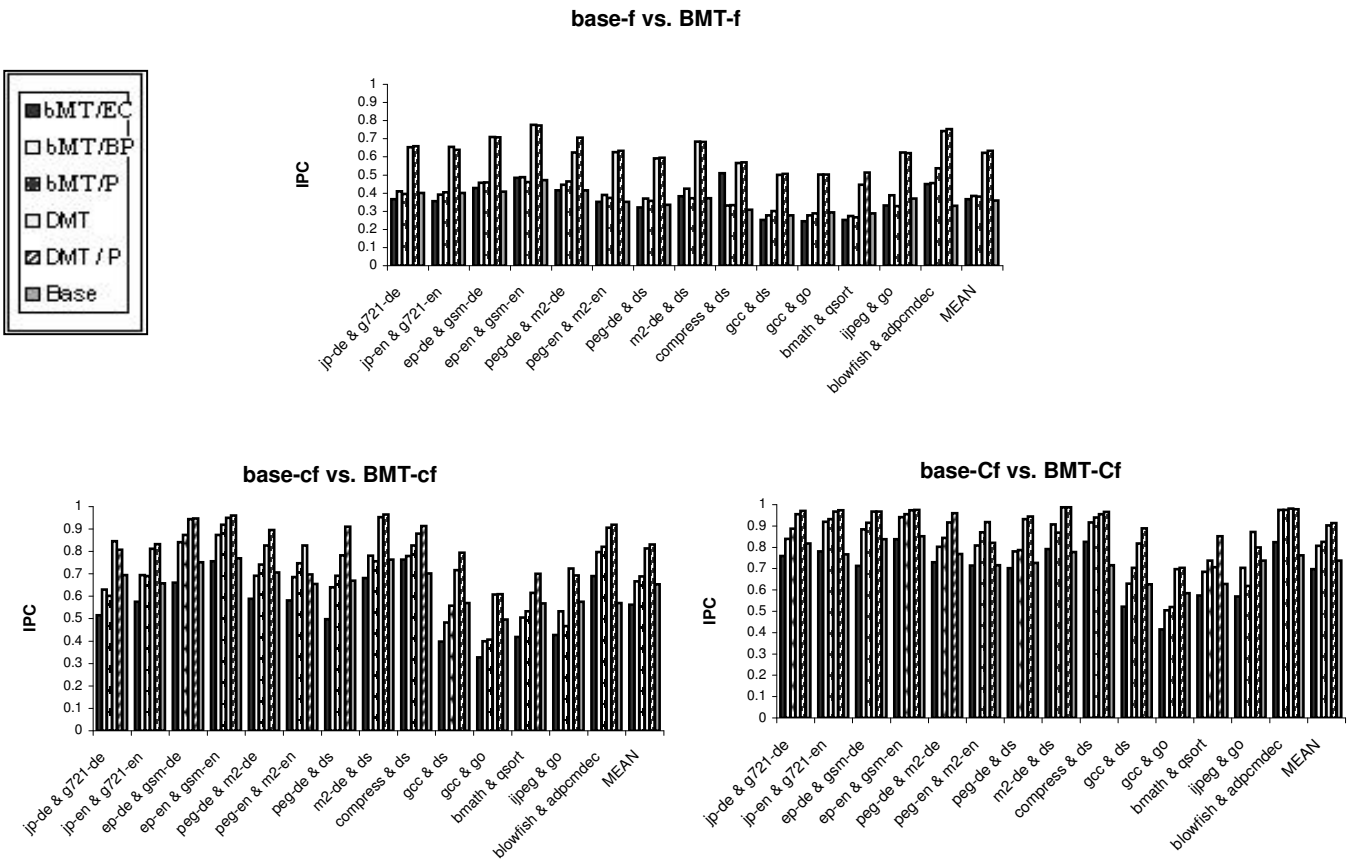


Figure 2: Comparison of baseline and MT configurations with similar organizations. BMT/EC, BMT/BP, and BMT/P represent every-cycle, blocked, and prioritized versions of BMT; dMT and dMT/P represent non-prioritized and prioritized versions of dMT; and Base represents the non-multithreaded processor.

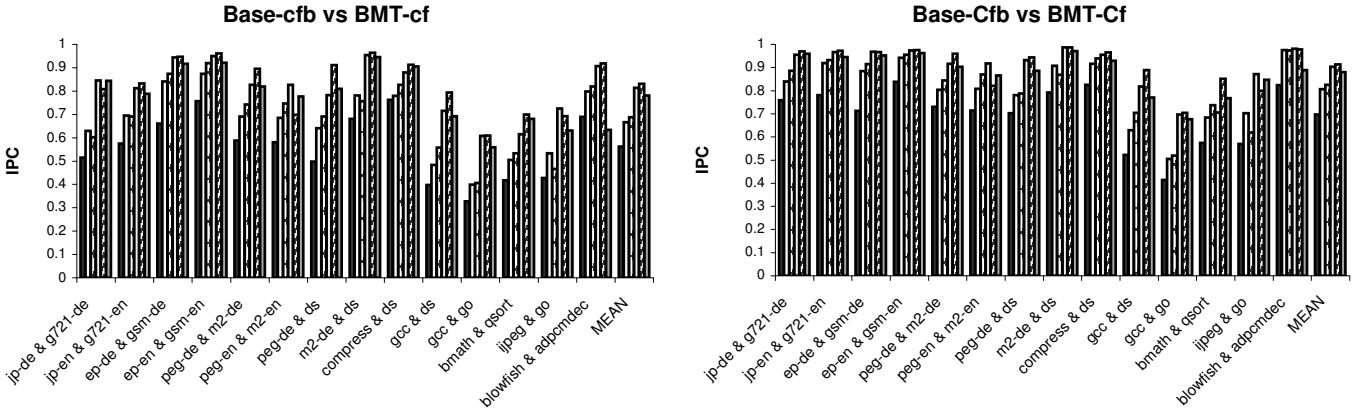


Figure 3: Comparison of baseline and MT configurations with similar organizations except lack of branch prediction for MT.

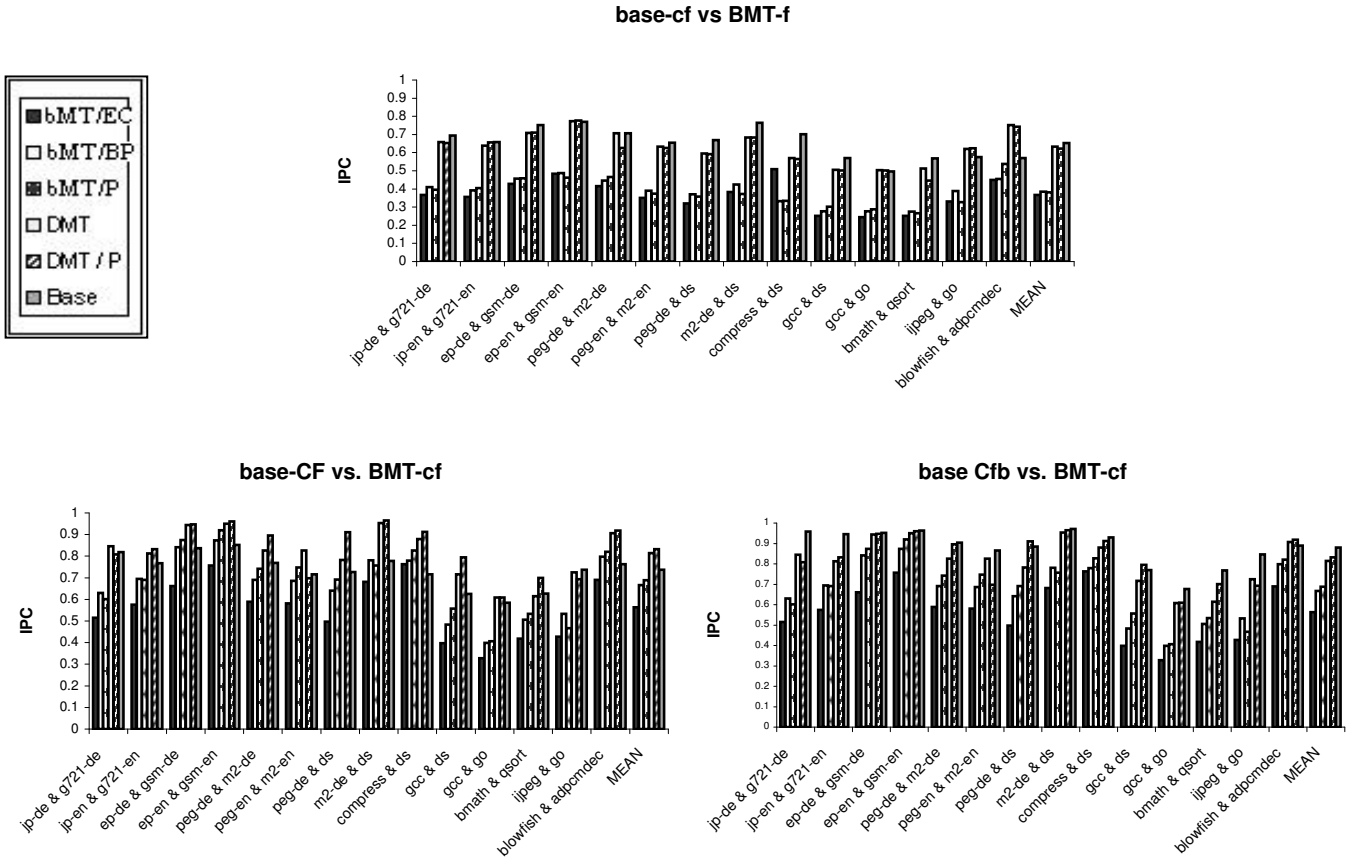


Figure 4: Comparison of baseline and MT configurations with similar organizations except that the MT configuration has less cache

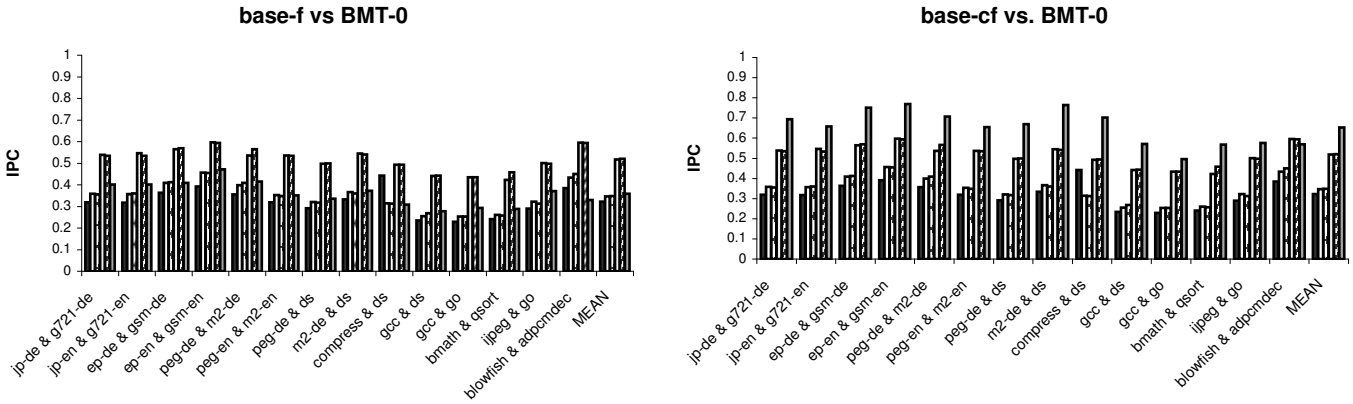


Figure 5: Comparison of baseline and MT configurations with similar organizations except that the MT configuration does no data forwarding.

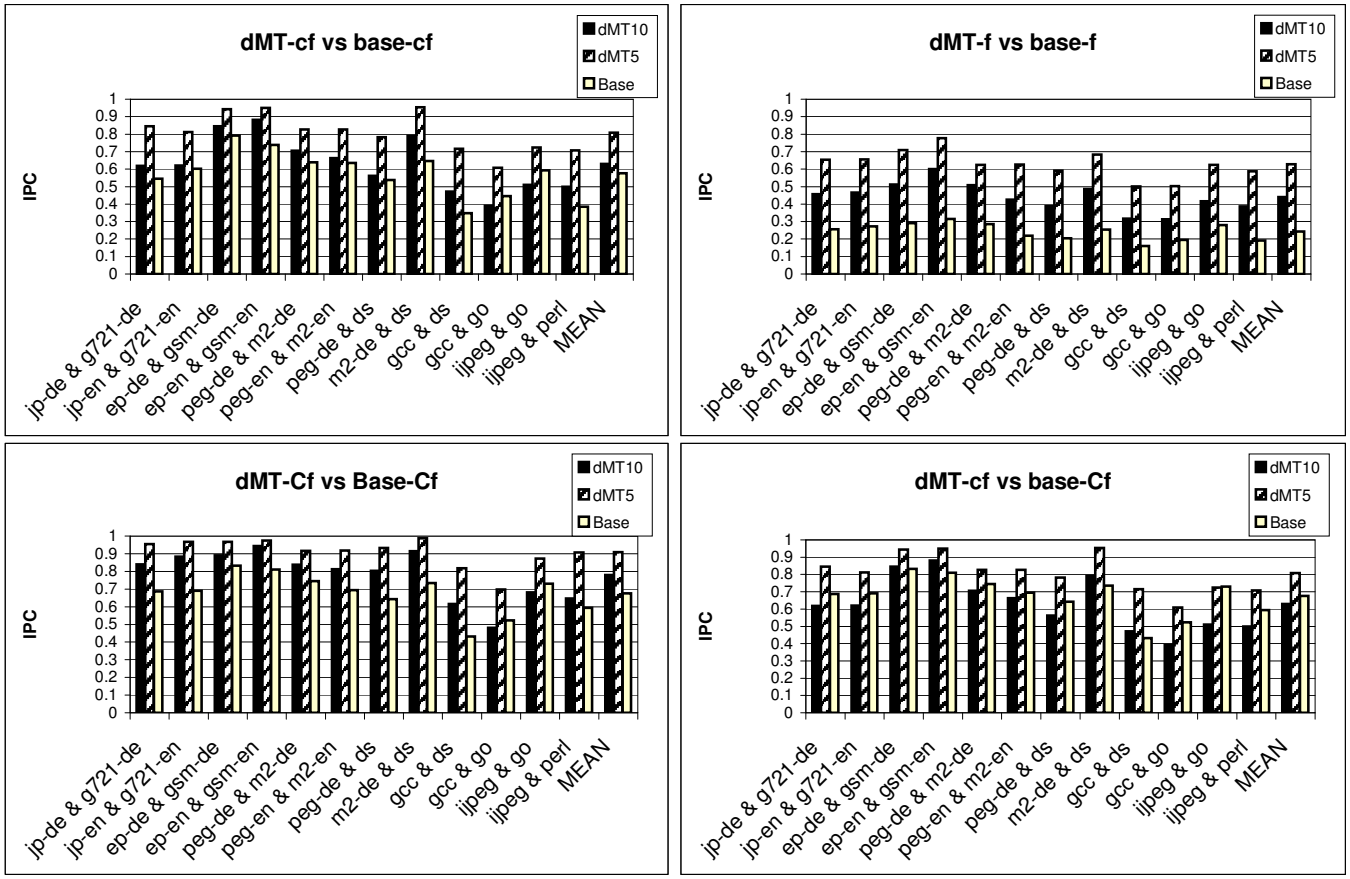


Figure 6: Comparison of Base and dMT configurations for larger cache miss penalty of 10 cycles; dmt5 is the performance of the 5 cycle miss penalty configuration from previous excerpts.

absence of branch prediction (which in any case is probably not desirable for workloads requiring determinism) and except for the possibility of cache contention between the dMT threads.

Note that, in the large-cache configuration, dMT comes quite close to the ideal IPC of 1, even though dMT omits a branch predictor. Also note that dMT outperforms BMT for all these configurations.

Among all BMT configurations, prioritized tends to be the best, followed by blocking, followed by every-cycle. This differs from what Farrens and Pleszkun found, namely that with a cache, blocking outperformed prioritized. The reason is that they run a benchmark pair until both programs complete. One program inevitably finishes first, and the other thread then experiences sequential, non-multithreaded execution. This situation is exacerbated in their measurements of BMT/Prioritized, because the prioritized program finishes even earlier, leaving the other to experience an even longer period of sequential execution. Our measurements, on the other hand, stop when one program finishes. Our rationale is that this gives a truer measure of BMT's and dMT's potential. Note that we treat both BMT and dMT the same.

Figures 3 and 4 compare different organizations that highlight the tradeoffs that multithreading permits. *We find that dMT can be used in place of a dynamic branch predictor or to allow substantially smaller caches.* Table 1 shows that dMT can also be used in place of forwarding, although this seems an unlikely design choice.

Of course, with only two threads, dMT is not able to recapture all stall cycles. This means that in the extreme cases we examine, where the MT configurations are substantially handicapped compared to the baseline, dMT's performance is outpaced by the baseline architecture. A few of these are shown in Table 1, but since the results are negative, we omit corresponding graphs in the interest of space. While dMT is not a cure-all, its ability to recapture stall cycles is impressive, and this paper suggests that it makes possible many interesting hardware tradeoffs.

Figure 3 shows that block multithreading can be used to replace a small, 128-entry, 2-bit bimodal predictor. For

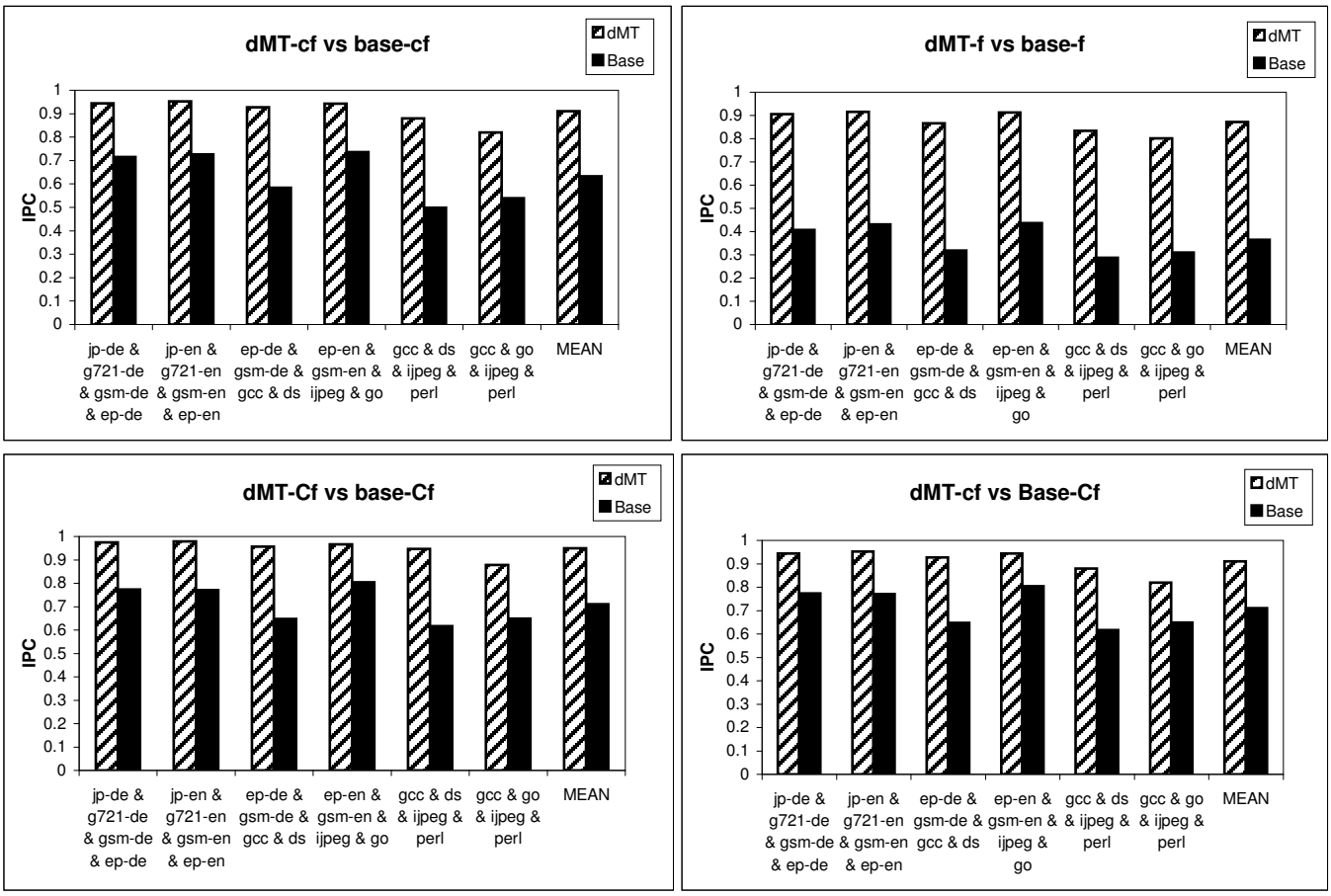


Figure 7: Comparison of Base and dMT configurations for running four thread contexts

*base-cfb* vs. *MT-cf*, dMT is superior for all but two benchmark pairs. For *base-Cfb* vs. *MT-Cf*, dMT is superior for all but one benchmark pair, *jpeg-decode & g721-decode*, which is a tie. This means that dMT is successful in finding useful work in the alternate thread when a branch is detected. From a hardware cost standpoint, let us assume that the branch predictor and register file both use minimum-size transistors [21]. Each register-file cell will still be twice larger than a branch predictor cell, due to the registers' extra read/write ports, so the extra register file is equivalent to 4096 branch-predictor bits. Even if each pipeline-register cell is four times larger than a branch-predictor cell, two-way dMT and the baseline end up almost the same in terms of hardware cost. Four-way dMT, however, will be somewhat more expensive, because the extra register files and pipeline registers will more than consume the hardware savings of eliminating the small branch predictor.

Figure 4 shows that dMT can be used to reduce cache sizes. *dMT-cf* is better than *base-Cf*—usually substantially so—for all but two benchmark pairs. This means that dMT can be used to reduce, without penalizing throughput, the cache configuration in an aggressive system like the ColdFire v4, with a 16-KB I-cache and 8-KB D-cache, both four-way associative, down to a more modest 8-KB I-cache and 2-KB D-cache, both two-way associative. Indeed, even after this reduction in cache, dMT's throughput is still 10.4% better. From a hardware cost standpoint, let us assume that the cache and register file use minimum-size transistors. The extra register file is equivalent to 4096 cache bits. But the reduction in cache size is 124,672 bits. Even when the extra register files and pipeline registers are accounted for, both two-way and four-way dMT are clearly still a substantial net win from a hardware standpoint.

From a throughput standpoint, neither dMT nor BMT can be used to replace the D-cache entirely, as seen in Table 1 for *base-cf* vs. *MT-f*, where dMT is 5% worse and BMT more so (but again, with no cache, BMT/every-cycle performs quite well). It remains to be seen whether a very tiny D-cache might suffice for dMT. It is also interesting to note in the

*base-Cfb* vs. *MT-cf* comparison in Table 1 that reducing both the I- and D-cache sizes and also removing the dynamic branch predictor incur only modest reductions in throughput for many of the benchmarks; on average, dMT is only 7.6% worse.

Figure 5 shows that a processor without D-cache can forego data forwarding and still achieve gains in throughput of 43% with dMT (and 25% with BMT/every-cycle), but a processor with cache cannot give up both the cache and the data forwarding together—BMT is now too severely handicapped, and throughput falls by 20.7%.

Adding the required multiplexor between the multiple pipeline registers required by dMT may slow the clock rate. Since adjusting clock rate to accommodate for these multiplexors is equivalent to lowering IPC for the same clock rate, the IPC improvements in Table 1 show how much change in clock rate can be accommodated. For example, if we assume that dMT will reduce the clock rate by ten percent, adding dMT instead of a branch predictor (*base-cfb* vs. *dMT-cf* or *base-Cfb* vs. *dMT-Cf*) may or may not be worthwhile, but even with a possible ten percent penalty due to extra hardware, dMT still improves throughput substantially for similar configurations (*base-f* vs. *dMT-f*, *base-cf* vs. *dMT-cf*, and *base-Cf* vs. *dMT-Cf*), and dMT allows use of smaller caches without reducing throughput (*base-Cf* vs. *dMT-cf*). If the multiplexing can be implemented with less overhead, other configurations may become competitive as well. Of course, it may be that even small reductions in throughput would be tolerated if enough hardware savings can be realized, for example the branch predictor (*base-cfb* vs. *dMT-cf* and *base-Cfb* vs. *dMT-Cf*).

To explore sensitivity to cache miss latency, Figure 6 shows the performance of similar base and dMT configurations, except that dMT is evaluated with both five- and ten-cycle miss penalties. The larger miss penalty still exhibits positive, albeit diminishing returns from dMT. This figure shows that dMT will maintain the same relative performance edge at least for modestly higher-latency L2 caches or for fast embedded DRAM.

Figure 7 gives the performance of dMT with four thread contexts. Here we formed quadruplets by selecting a mix of threads for which we observed varying performances in the two-thread experiments. These results show that running a high number of threads can come quite close to masking the penalty of an L1 cache absence, simply by switching the active thread as necessary. There is even less need for cache for this configuration, since the large cache configuration does not outperform the small cache configuration by a reasonable margin. Of course, there are other ways to run the same four threads, for example two dMT processors each running two threads. This will give higher per-thread performance but exhibit more hardware cost. This illustrates the range of interesting tradeoffs that dMT opens up, an extensive study of which is another interesting question for future work.

## 6 Conclusions and Future Work

This paper presents differential multithreading, an inexpensive technique for sharing a single pipeline between multiple active threads. Our contribution is the addition of duplicated pipeline registers which enable the capture of in-flight instructions anywhere in the pipeline. This organization is thus able to respond not only to data hazards and branch delays, but also to misses in the primary I-cache and D-cache. This combines beneficial aspects of block multithreading [2] and Runahead [4], and allows attractive hardware tradeoffs.

We went on to show that dMT can reclaim a significant amount of wasted cycles. For processors without data caches, dMT boosts throughput by 71.6% over a non-multithreaded organization, and for processors with cache, dMT boosts throughput by 23–24%. Better yet, instead of using dMT to increase the throughput of more complex, single-threaded configurations, it can be used—*without reducing throughput*—to eliminate the dynamic branch predictor or to reduce instruction- and data-cache sizes.

Our results also show that dMT is consistently superior to BMT, because dMT handles a wider variety of stall conditions. Furthermore, like the original BMT concept, our design can take advantage of each of the thread switching policies. Of particular note is the prioritized switching policy, which returns control to the prioritized thread immediately after its stall condition is resolved, making dMT viable in real-time systems.

Our results suggest a variety of interesting avenues for future work. Incorporating and evaluating branch prediction in a dMT pipeline, and evaluating multi-threaded, data-sharing programs are both interesting questions. It would also be interesting to evaluate the benefits of dMT from an energy-efficiency standpoint. Finally, dMT opens up a new dimension of design, allowing CPUs with various degrees of dMT to be combined into a multiprocessor system, and the proper balance of dMT with the number of processors is an open question.

**Acknowledgments**

This material is based upon work supported in part by the U.S. National Science Foundation under grant. nos. CCR-0082671 and CCR-0133634. We would also like to thank Doug Clark, Yingmin Li, and Mircea Stan for their valuable advice and assistance as we developed this work and the anonymous reviewers for their detailed and insightful comments.

## References

[1] K. R. Hirst, J. W. Haskins Jr., and K. Skadron. dMT: Inexpensive throughput enhancement in small-scale embedded microprocessors with differential multithreading. *IEE Proceedings on Computers and Digital Techniques*, Jan. 2004. To appear.

[2] M. K. Farrens and A. R. Pleszkun. Strategies for achieving improved processor throughput. In *Proc. ISCA-18*, pages 362–369, May 1991.

[3] J. W. Haskins, Jr., K. R. Hirst, and K. Skadron. Inexpensive throughput enhancement in small-scale embedded microprocessors with block multithreading: Extensions, characterization, and tradeoffs. In *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, pages 319–28, Apr. 2001.

[4] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. of the 1997 ACM International Conference on Supercomputing*, pages 68–75, Jul. 1997.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001.

[6] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating multimedia and communications systems. In *Proc. of the 30th International Symposium on Microarchitecture*, pages 330–35, Dec. 1997.

[7] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. http://www.specbench.org/osg/cpu95, Dec. 1999.

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

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

[10] 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.

[11] 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.

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

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

[14] L. Gwennap. ARM10 points to set-tops, handhelds. *Microprocessor Report*, Nov. 16 1998.

[15] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, pages 23–29, Jul.–Aug. 1999.

[16] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[17] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, Oct. 1994.

[18] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. http://www.specbench.org/osg/cpu2000, Dec. 1999.

[19] 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 Trans. Computers*, 48(11):1260–81, Nov. 1999.

[20] J. Turley. ColdFire doubles performance with v4. *Microprocessor Report*, Oct. 26 1998.

[21] M. Steinhaus, R. Kolla, J. Larriba-Pey, T. Ungerer, and M. Valero. Transistor count and chip-space estimation of simplescalar-based microprocessor models. In *Proc. of the Workshop on Complexity-Effective Design*, June 2001.