Inexpensive Throughput Enhancement in Small-Scale Embedded Microprocessors with Block Multithreading: Extensions, Characterization, and Tradeoffs

John W. Haskins, Jr. Kevin R. Hirst Kevin Skadron
University of Virginia, Department of Computer Science
Charlottesville, VA 22904
{predator,krh2n,skadron}@cs.virginia.edu

Abstract

This paper examines differential multithreading (dMT) as an attractive organization for coping with pipeline stalls in small-scale processors like those used in embedded environments. The paper proposes extensions to block multithreading to cope with data- and instruction-cache misses, and then explores some of the design tradeoffs that this enables. Results show that dMT boosts throughput substantially and can in fact replace *dynamic branch prediction or data forwarding, or can be used to reduce the sizes of the instruction and data caches.*

Block multithreading, described by Farrens and Pleszkun, is a technique to achieve high throughput from a single-issue microarchitecture by switching among multiple instruction streams in response to pipeline stalls. Although single-issue organizations are no longer used in high-performance processors, they remain common even in newly-designed processors for small-scale, embedded devices. Like the original description of block multithreading, dMT uses auxiliary pipeline registers to save the state of in-flight instructions. By coping with data- and instruction-cache misses, however, our implementation can attack all the major sources of pipeline stalls. Overall, we find that dMT can substantially lower the cost and complexity of microprocessors for embedded environments, especially environments for which throughput rather than speed is the primary concern. In addition, dMT is an attractive prospect for use in chip-multiprocessing environments.

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. *Block multithreading* (BMT) is a relatively 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; [1] this technique was first described by Farrens and Pleszkun in [6]. We extend BMT by adding the ability to switch hardware contexts in response to misses in the instruction- and data-cache and call this organization *differential multithreading* (dMT). Note that BMT and dMT differ from *simultaneous multithreading* (SMT) [12, 20], which requires multiple functional units and therefore has no single-pipeline implementation.

Benefits. If the two threads can entirely hide each others' stalls, the pipeline's maximum throughput of one instruction per cycle (IPC) can be attained, meaning that 100% of CPU cycles are spent on useful computation with no cycles wasted on stalls. Conventional pipelines fall short of the maximum 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, data forwarding) which increase pipeline utilization for only a single instruction stream.

Although single-issue organizations are no longer used in high-performance processors, they remain common even in newly-designed processors for small-scale, embedded devices. Some of these processors are in fact cacheless, 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 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

[1] The terms *context*, *instruction stream* and *thread* are used interchangeably throughout this paper. In particular, when we refer to a "context-switch", we mean a pipeline switch between its active instruction streams and not an OS-level switch among processes.

Copyright © 2001 IEEE. Published in the Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference (IPCCC 2001), April, 2001, Phoenix, Arizona. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

on throughput of multiple, semi-independent applications.

The largest benefit of differential multithreading, which benefits all the above scenarios, is its ability to increase throughput so effectively that other processor structures, like caches or branch predictors, can be made smaller or even eliminated. For example, section 5 shows that when dMT with neither forwarding nor D-cache is compared against the non-multithreaded baseline architecture with forwarding but no D-cache, dMT improves IPC significantly on all pairs of benchmarks. This means that differential multithreading can be used in lieu of forwarding. Similarly, we find that dMT can be used to reduce cache size and associativity or as a way to eliminate branch prediction.

Contributions. Our contribution stems from our extensions to block multithreading, which can be thought of as combining aspects of the BMT [6] and Runahead [5] 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 both MediaBench [11] and SPECint95 [19]. Our paper extends preliminary results from [10].

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 [6]. It describes three different policies for interleaving instructions: *every-cycle*, *blocked* and *prioritized*. Every-Cycle changes contexts after every clock cycle, and Blocked only in response to stalls. For prioritized, the context with priority resumes execution as soon as its stall condition resolves, regardless of the status of the other context; this policy is useful for real-time development. By resuming its execution as soon as its stall condition resolves, the context with priority will incur no penalty beyond what it would if it were executed alone, thus preserving the predictability of its execution time.

Traditional multithreaded architectures such as the Tera [2] 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 access latency and permits the Tera to be completely cacheless.

Simultaneous multithreaded (SMT) architectures [12, 20] 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.

Neither the Tera nor SMT is readily deployable into an embedded system that needs the best possible performance and yet is still constrained by cost, size and power. Microprocessors for such environments cannot yet afford these types of hardware multithreading; they require too many hardware contexts, too much issue capability and too many functional units.

Three more systems that are related to block multithreading are APRIL [1], Runahead [5], and the use of informing-memory operations [13]. Like dMT, APRIL context switches on a cache miss; however, APRIL uses a more heavyweight context 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 squashed; 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 due to data hazards, control 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 [15], used in Palm Pilots and the Zilog Z80 [16], used in Nintendo Gameboys. In addition, this simplicity means that techniques like out-of-order issue, dynamic branch prediction and large, highly associative caches may not be available. dMT uses a lightweight version of multithreading to squash these pipeline bubbles, thereby achieving better utilization and improving system throughput.

We extend prior work by:

- enabling block multithreading to context switch in response to I-cache and D-cache misses
- showing that block multithreading can be used in place of a cache, forwarding or dynamic branch prediction
- simulating both BMT and dMT on more modern, representative benchmarks (the original BMT work used only Livermore Loops)
- briefly considering the possible role of differential multithreading in a chip multiprocessor

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 [8].

3.1 Pipeline Organization

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 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[2] 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. The dMT pipeline organization is depicted in Figure 1. Note that, if precise exceptions are not important, then the EX-MM register need not be duplicated; independent instructions could be routed around a cache miss directly to the WB stage.

Each data and control signal entering the duplicated pipeline registers has a fanout of two: one copy into each "half" of the pipeline register. Each resulting 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 context 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 our dual-threaded dMT model, if neither thread's stall condition is resolved, the pipeline experiences a true stall until one thread or the other can proceed.

3.2 Sources and Treatment of Stalls

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 continue to flow through the pipeline unhindered; instructions in the dependent part are captured 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.

[2]Pipeline registers are used to latch data and control signals between stages.

Data Hazards. Result forwarding eliminates data hazards by using a bypass network to route results to dependent instructions as soon as they are available. The required bypass networks are simple for processors like our five-stage pipeline. More deeply pipelined processors, however, require more sophisticated bypassing networks. dMT can reduce or eliminate the need for bypassing by substituting computation from the alternate thread.

If data forwarding is not implemented, the committable part of the instruction stream consists of the instructions in the EX, MM and WB stages of the pipeline; the dependent part consists of the instructions from IF and ID. Once interlocking hardware detects a data hazard, the instructions in IF and ID are frozen. When control of the pipeline is returned to these 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 register read takes place in ID.

This paper explores configurations both with and without data forwarding. In simulations that include forwarding, data hazards receive no special treatment. In those simulations without forwarding, we assume that data hazards are detected in the ID stage by the same interlocking hardware that the ARM10 uses to implement forwarding.

Control Hazards and Dynamic Branch Prediction. While not all embedded microprocessors have dynamic branch prediction, almost all processors implement some form of branch prediction; at the minimum, processors continue fetching sequentially past branches, and this is the same as statically predicting branches to be not taken. Without prediction, the processor must stall every time it encounters a branch. Even with prediction, mispredictions impose a penalty in squashed instructions. The misprediction penalty grows as the pipeline becomes longer, so the ten-stage ColdFire v4 [21], for example, pays a heavy penalty for each misprediction and actually uses dynamic prediction. Dynamic branch predictors, however, are large, complex, and power-hungry structures that may not be feasible in tightly constrained systems.

In our experiments, we always simulate a five-stage pipeline. We explore configurations with and without branch prediction, and explore the ability of dMT to replace the small dynamic branch predictors that would be used in non-multithreaded embedded processors. When a dynamic predictor is used, it is a 2-bit bimodal predictor [18] with a 128-entry branch target buffer. If no dynamic predictor is used, the processor implements static-not-taken prediction, fetching sequentially past branches. In either case, if a branch is incorrectly predicted, this will be detected at the conclusion of the EX stage, causing the instructions in the IF and ID stages to be squashed and incurring a two-cycle misprediction penalty.

dMT on the other hand treats every conditional branch and indirect branch as a control hazard and immediately freezes the active thread, switching execution to the alternative context. Detecting control hazards requires predecode logic that can identify conditional and indirect

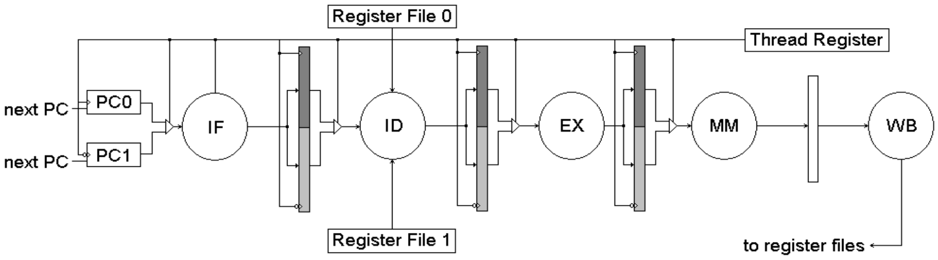

Figure 1: The dMT pipeline.

branch instructions in the IF stage or the presence of appropriate pre-decode bits in the instruction cache. The committable part after a control hazard therefore consists of every instruction before and including the offending branch; the dependent part is "empty" because the IF stage will not fetch further instructions from this context until the branch completes in the EX stage—no state need be snapshotted.

Instruction-Cache Misses. As with D-cache misses, we assume a fixed latency of five cycles for I-cache misses. When the IF stage misses in the instruction cache, the active context is frozen and the alternate thread takes over. The committable part consists of all instructions currently in the pipeline, so no state from the active context need be captured. If the context which is inactive at the time of the miss contains an instruction in its half of the IF-ID pipeline register, this instruction is promoted to the ID stage when this context resumes, while the IF stage simultaneously begins fetching the next instruction. An I-cache miss therefore need not introduce a bubble into the pipeline.

Data-Cache Misses. In a data cache miss, the dependent part 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 controller detects a cache miss, dependent instructions are frozen in their respective half of the pipeline registers. We assume a fixed cache-miss latency of five cycles, a reasonable latency for an on-chip backing store that might be found in system-on-a-chip implementations. We also assume that all memory references in a cacheless system take five cycles, once again assuming that the system memory resides on-chip.

We model three different cache configurations: a larger configuration with a 16-KB I-cache and an 8-KB D-cache, both 4-way associative and with 32-byte lines; a smaller configuration with an 8-KB I-cache and a 2-KB D-cache, both 2-way associative; and a configuration with a 2-way associative, 8-KB I-cache and no D-cache. The first is based on the Motorola ColdFire v4 [9], and the second on the SparcLite 86934 [7]. The final configuration is used to model systems with no data cache. One final note: keeping different contexts' memory references from corrupting each other is handled by conventional caching techniques. Virtually-indexed caches require a process identifier for each cache line; physically indexed caches automatically avoid protection violations.

3.3 Implementation Issues

To summarize, dMT requires the addition of duplicated IF-ID, ID-EX and EX-MM pipeline registers; an extra PC register; an extra register file; a thread register; fanout of two for the inputs to the IF-ID, ID-EX and EX-MM pipeline registers; fanin of two for the inputs to the ID, EX and MM stages, with multiplexors to select the appropriate signals; and pr-decode bits in the I-cache (or pre-decode logic in the IF stage) to detect control hazards. 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 input multiplexors to the ID, EX and MM 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. On the other hand, dMT recaptures stall cycles in one thread with useful execution from the other thread. This means that stalls that result from a smaller cache or even the absence of a cache, a branch predictor, or a bypassing network do not necessarily reduce throughput. dMT can therefore be used as a general-purpose replacement for these other structures.

4 Simulator and Benchmarks

4.1 Simulator

We model BMT and dMT by building a cycle-accurate simulator upon *sim-cache* from release 3.0a of the University of Wisconsin's *SimpleScalar* software package [3] to model a simple 5-stage, in-order-issue pipeline with multithreading capabilities. The result, called *sim-dmt*, can model either a conventional, single-issue processor, or a BMT or dMT processor running multiple contexts. We assume that in the absence of other stalls, all instructions take one cycle to execute.

This paper focuses on showing how much dMT can boost throughput, and the extent to which dMT can be used to eliminate cache, branch prediction and bypassing. Unfortunately, the cost of our implementation's extra hardware and the cost savings that it permits are so implementation-dependent that a detailed comparision is beyond the scope of this paper. Instead, we illustrate potential savings by merely counting the difference in bits between the dMT and non-multithreaded configurations (see

Table 2). Naturally, this does not include the cost of dMT's extra multiplexors, nor dMT's potential savings in bypassing hardware.

4.2 Benchmarks

For our experiments, we used a mix of benchmark programs from the SPEC95 [19] and MediaBench [11] suites, and we also used the Dhrystone benchmark [22]. Rather than show data averaged across all benchmarks as in [6], 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 quite a bit for benchmarking embedded microprocessors and because for better or worse it is well understood.

Benchmark	Abbreviation	Fast-forward insts
go	go	925 M
gcc (cc1)	gcc	220 M
compress	compress	2575 M
ijpeg	ijpeg	823 M
perl	perl	600 M
mpeg2encode	m2-en	175 M
mpeg2decode	m2-de	100 M
g721encode	g721-en	200 M
g721decode	g721-de	205 M
gsmencode	gsm-en	100 M
gsmdecode	gsm-de	100 M
epicencode	ep-en	*
epicdecode	ep-de	*
jpegencode	jp-en	*
jpegdecode	jp-de	*
pegwitencode	peg-en	*
pegwitdecode	peg-de	*
dhrystone	ds	*

Table 1: Benchmark summary.

"Fast-forward insts" indicates the length of the preliminary phase of simulation before statistics-gathering; "*" denotes very short benchmarks which were run to completion with no fast-forwarding. The Abbreviation column give abbreviations for the benchmark names used in the graphs of figures 2, 3, 4, and 5. For benchmarks that are not run to completion, statistics were gathered for 100 million instructions. SPEC benchmarks use the reference inputs.

Except for *epic, jpeg, pegwit* and *dhrystone*, which are short and were run to completion, all simulations were fast-forwarded according to the methodology in [17] in order to avoid unrepresentative initial behavior, and statistics were gathered for the next 100 million instructions. Fast-forward distance appears in Table 1.

The benchmarks were compiled with gcc 2.6.3 (the SimpleScalar compiler) and -O3 optimization for the SimpleScalar PISA instruction set. Simulations include all references to libraries, but do not model system calls—SimpleScalar performs system calls by proxy. An important caveat for experiments that use neither BMT nor bypassing is that the compiler was unaware of the lack of a forwarding network and therefore made no attempt to schedule instructions accordingly.

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 (16-KB I-cache and 8-KB D-cache, both 4-way)
c:	small cache configuration (8-KB I-cache and 2-KB D-cache, both 2-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.

We present data for a total of 14 different pairings of 18 different benchmark programs. We used these pairs to make 10 different comparisons of non-multithreaded and multithreaded processors. For each of the 10 comparisons, 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.). Figure 3 adds the ColdFire's dynamic branch predictor to each baseline configuration, but for each of these comparisons, MT still omits branch prediction, context switching instead. This demonstrates MT's robustness against control hazards. Figure 4 gives each of the baseline systems more cache than the MT systems. Finally, Figure 5 gives each baseline system data forwarding, while omitting forwarding from the MT systems. Note that some configurations appear in more than one comparison; this is in order to illustrate the tradeoffs multithreading permits. Table 2 gives the average improvement in throughput (IPC) for each comparison. In each figure, we evaluate the performance of the same three context switching policies evaluated by Farrens and Pleszkun (every cycle (BMT/EC), blocked (BMT/BP), and prioritized (BMT/P)); a non-prioritized and a prioritized version of dMT (dMT and dMT/P); and the baseline, non-multithreaded organization (Base).

Baseline Config.	MT Config.	Graph	Δ IPC
base-f (1,237)	MT-f (2,474)	Fig. 2	71.6%
base-cf (91,093)	MT-cf (92,330)	Fig. 2	23.4%
base-Cf (213,461)	MT-Cf (214,698)	Fig. 2	22.9%
base-cfb (99,285)	MT-cf (92,330)	Fig. 3	3.3%
base-Cfb (221,653)	MT-Cf (214,698)	Fig. 3	3.0%
base-cf (91,093)	MT-f (2,474)	Fig. 4	-5.0%
base-Cf (213,461)	MT-cf (92,330)	Fig. 4	9.5%
base-Cfb (221,653)	MT-cf (92,330)	Fig. 4	-8.3%
base-f (1,237)	MT-0 (2,474)	Fig. 5	42.6%
base-cf (91,093)	MT-0 (2,474)	Fig. 5	-21.7%

Table 2: Baseline vs. MT comparisons. The rightmost column gives the average improvement in throughput. Each configuration includes in parentheses the number of bits in the pipeline registers, cache, and branch predictor. These numbers of course do not include the extra multiplexors required by multithreading, nor the savings in bypassing hardware achieved by MT-0.

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 (non-prioritized) switches contexts. Arithmetic means taken across our set of benchmark pairs are reported in Table 3. As expected, with no forwarding (the "0" configuration), context switches are frequent (every 1.7 instructions), and as the configuration becomes more aggressive, context 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 3: Instructions between dMT instruction-stream switches (arithmetic mean).

5.2 Discussion

5.2.1 Performance of dMT

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. It boosts throughput by 71% in a processor with no data cache, by slightly more than 23% in the small-cache configuration, and by slightly less than 23% in the large-cache configuration. For dMT, sometimes the conventional policy outperforms the prioritized policy and vice-versa. Overall, the difference is small, although specific benchmarks experience larger differences. This is due to specific reactions to cache contention. Note that, in the large-cache configuration, dMT comes quite close to the ideal IPC of 1, even though dMT omits a branch predictor.

dMT outperforms BMT for all these configurations, except with the "Cf" and "cf" configurations for *gcc & go* and *ijpeg & go*. In this case, BMT with a "blocked" policy outperforms all alternatives. This is because these pairings suffer severe cache contention, and the BMT blocking configuration does not switch away on a cache miss (whereas BMT's every-cycle and prioritized schemes do switch away from a cache miss; prioritized switches away from a cache-miss stall if the prioritized thread is ready to run). Not switching away on a cache miss prevents the other thread's missing on the same cache location and evicting data that the first thread still needs. Note that these two unusual cases distort the means for these configurations.

Among BMT configurations (except for the aforementioned unusual cases), BMT with an every-cycle policy is substantially better than the BMT alternatives for the "f" configurations. The chief reason is that every-cycle makes some progress on one thread when the other is stalled on a cache miss. For the configurations with cache ("Cf" and "cf"), 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 or completes its alloted simulation length. 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, 4, and 5 compare different organizations that highlight the tradeoffs that multithreading permits. *We find that dMT can be used in place of forwarding or a small cache or a dynamic branch predictor.* Of course, with only two contexts, 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. This does not, however, diminish dMT's value. While it is not a cure-all, its ability recapture stall cycles is impressive, and these data suggest that it does make possible many interesting hardware tradeoffs.

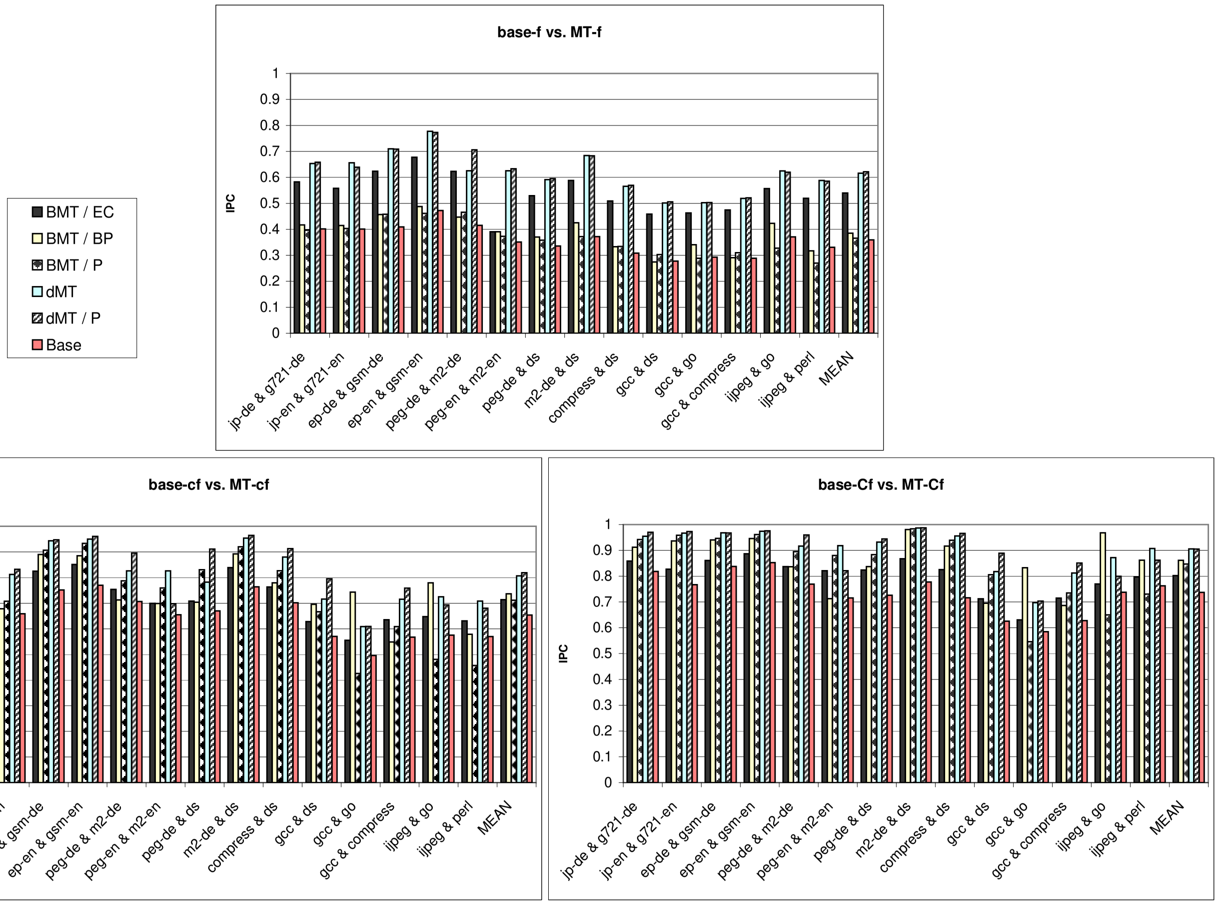

Figure 2: Comparison of baseline and MT configurations with similar organizations. BMT/EC, BMT/BP, and BMT/P represent every-cycle, blocked, and prioritized versions of BMT; dMT and dMT/P represent non-prioritized and prioritized versions of dMT; and Base represents the non-multithreaded processor.

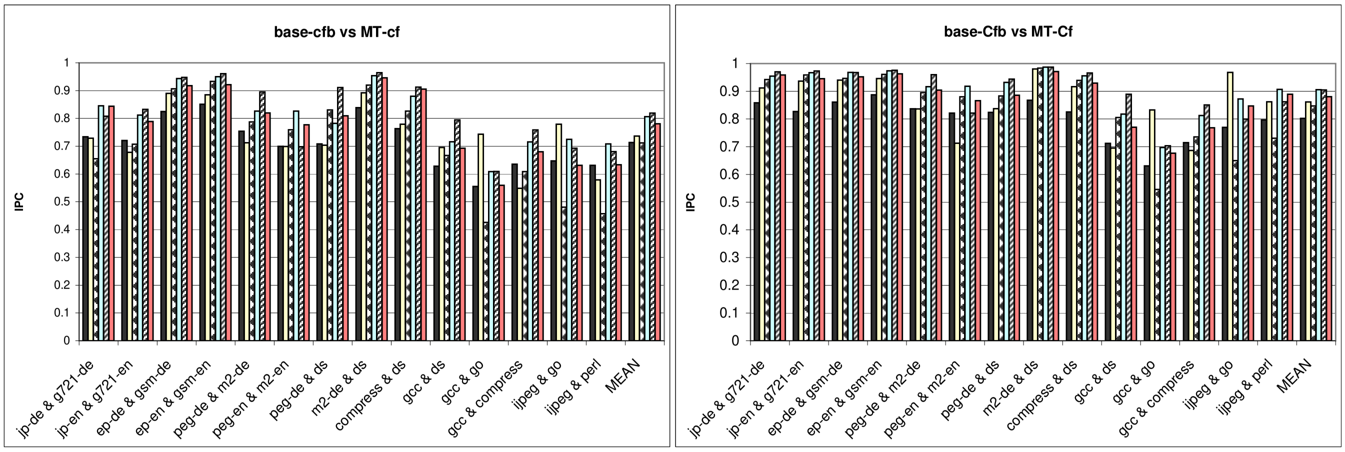

Figure 3: Comparison of baseline and MT configurations with similar organizations except lack of branch prediction for MT.

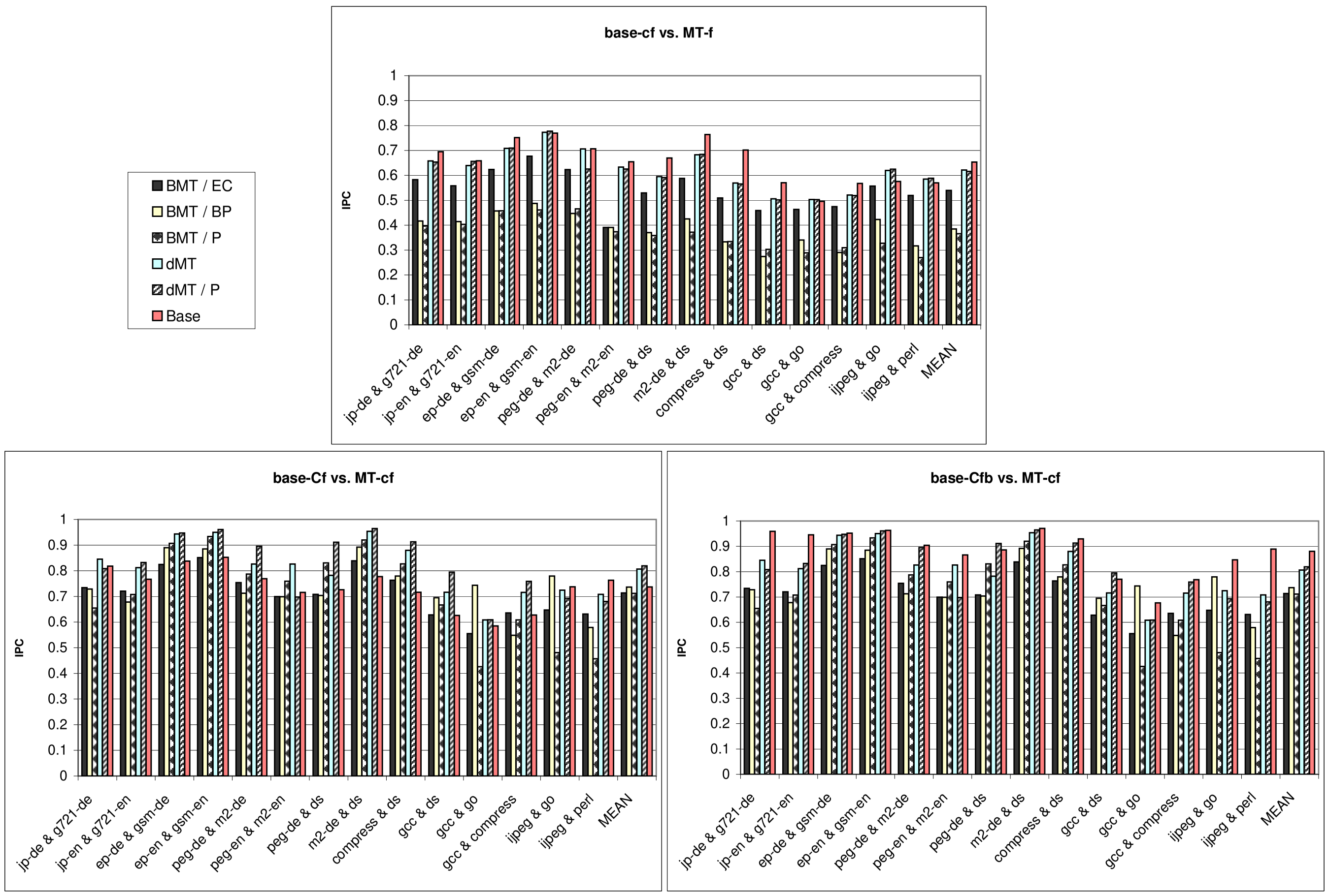

Figure 4: Comparison of baseline and MT configurations with similar organizations except that the MT configuration has less cache.

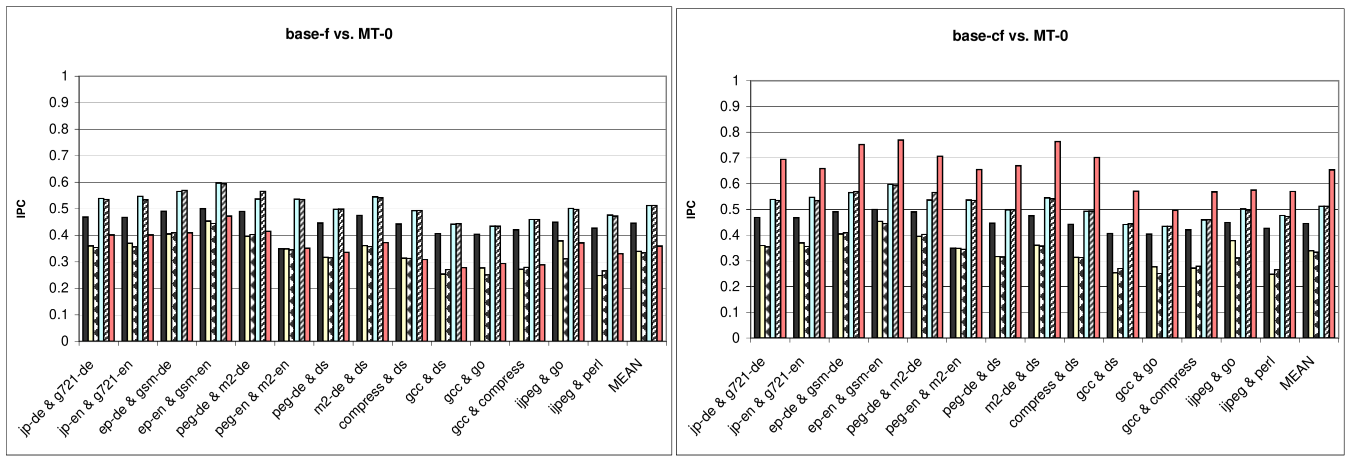

Figure 5: Comparison of baseline and MT configurations with similar organizations except that the MT configuration does no data forwarding.

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

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 benchmarks (both involving *ijpeg*). This means that dMT can be used to reduce, without penalizing throughput, the cache configuration in aggressive system like the ColdFire v4, with a 16-KB I-cache and 8-KB D-cache, both 4-way associative, down to a more modest 8-KB I-cache and 2-KB D-cache, both 2-way associative. Indeed, even after this reduction in cache, dMT's throughput is still 9.5% better.

On the other hand, neither dMT nor BMT can be used to replace the D-cache entirely, as seen in the comparisons between *base-cf* and *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–MT-cf* comparison that reducing both the I- and D-cache sizes and also removing the dynamic branch predictor produce only modest reductions in throughput for many of the benchmarks; on average, dMT is only 8.3% worse. Nevertheless, asking dMT or BMT to not only reduce cache size, but to simultaneously accommodate the elimination of a branch predictor, is too much to ask.

Finally, 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 21.7%.

5.2.2 CMP Issues

In addition to measuring dMT's performance against alternative single-issue processor cores, which is of interest for small-scale processors intended for low-cost embedded environments, we briefly explored the potential of dMT in a single-chip multiprocessor (CMP) environment [14]. CMP systems integrate multiple processor cores onto a single chip. Each conventional core has its own primary caches, branch predictor, store queue, etc. and shares a single, unified secondary cache with all the other processor cores.

A CMP system made with dMT cores, however, can forego some of these structures. For this reason, we compared a CMP system with three dMT cores against a CMP system with two conventional, dual-issue cores.

The dMT chip multiprocessor (CMP-dMT) simulations simulated three dMT cores with data forwarding, an 8k, 2-way associative instruction cache and an 8k, 4-way associative data cache; all three cores shared a unified 512k, 4-way associative secondary cache. The other CMP simulations simulated two conventional, dual-issue, non-BMT, in-order cores (CMP-base) with data forwarding, a 2048-entry bimodal branch predictor with a 512-entry branch-target-buffer and 8-entry return-address-stack, a 4-entry store queue, an 8-entry instruction queue and caches identical to CMP-dMT. These dual-issue cores are modeled after the Alpha 21064 [4], but with dual integer issue capacity instead of one integer, one FP; and with dual cache ports. According to our very crude bit-counting measure, three dMT cores require fewer bits than two conventional dual-issue processors, because the BMT cores can forego branch prediction and some other small structures. A further consideration is that the dual register files per dMT core are most likely less expensive than the one multi-ported register file per conventional dual-issue core.

We used six of the SPEC integer benchmarks to perform our simulations. On CMP-dMT, all six benchmarks were loaded at once, two per core. CMP-base, on the other hand, must run only two benchmarks at a time. This is because the CMP-base system has only two cores, and each of those can only run one program at a time. In other words, while CMP-dMT can run six benchmarks in parallel (but any one benchmark can achieve 1 IPC at best, and maximum aggregate throughput is 3 IPC), CMP-base must run the benchmarks two at a time and requires three passes (but any one benchmark can potentially achieve 2 IPC due to dual issue, and the maximum aggregate throughput is 4 IPC).

On both systems, we measured mean throughput for all fifteen possible ways to group these six benchmarks into pairs. It turns out that the dual-issue cores barely achieve IPCs above 1, so the aggregate IPC for a two-core CMP-base system is barely above 2. And we found that the worst CMP-dMT result (IPC = 2.11) outperforms the best CMP-base result (IPC = 2.04)!

Of course, it is impossible to determine whether three single-issue dMT cores would have the same cost as two conventional, dual-issue cores, or whether they would have comparable cycle times, without a detailed implementation. Although a detailed implementation is beyond the scope of this work, it is indeed likely that a single-issue dMT core will be smaller than a conventional dual-issue core. The dMT core does require two register files and multiplexors on the inputs to each pipeline stage, but avoids the more sophisticated scheduling logic, the extra bypassing, and the expensive, multi-ported register file needed for dual-issue. Given these factors, the comparison of three single-issue dMT cores against two conventional, dual-issue cores is of interest, and dMT acquits itself well.

6 Conclusions

This paper presents differential multithreading, which builds upon block multithreading presented by Farrens and Pleszkun in [6]: an inexpensive technique for sharing a single pipeline between multiple hardware contexts. 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 BMT and Runahead [5], and allows dMT the possibility for attractive hardware tradeoffs.

We went on to show that dMT can reclaim a significant amount of wasted cycles. For cacheless processors, dMT boosts throughput by 71.6% over a non-multithreaded organization, and for processors with cache, dMT boosts throughput by 23%.

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 the data-forwarding, or to reduce instruction- and data-cache sizes. For example, dMT with an 8-KB I-cache, 2-KB D-cache, and only 2-way associativity boosts throughput by 9.5% over a conventional processor with a much larger 16-KB I-cache, 8-KB D-cache, and 4-way associativity!

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 context switching policies. Of particular note is the prioritized switching policy, which returns control to the prioritized context immediately after its stall condition is resolved, making block multithreading viable in real-time systems.

Finally, we explored whether block multithreading might allow the use of less-expensive cores in a chip-multiprocessing environment [14]. It seems likely that a single-issue dMT core should be less costly than a conventional, dual-issue core. We therefore compared a CMP system with three, single-issue dMT cores against one with two conventional, dual-issue cores, and found that the dMT system performed slightly better.

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 Supercomputing*, May 1990.

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

[4] Digital Semiconductor. *DECchip 21064/21064A Alpha AXP Microprocessors: Hardware Reference Manual*, June 1994.

[5] 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.

[6] M. K. Farrens and A. R. Pleszkun. Strategies for achieving improved processor throughput. In *Proc. of the 18th International Symposium on Computer Architecture*, pages 362–69, May 1991.

[7] C. P. Feigel and M. Enfield. Fujitsu extends SparcLite family. *Microprocessor Report*, Jun. 20 1994.

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

[9] T. R. Halfhill. Motorola thaws ColdFire v4. *Microprocessor Report*, pages 49–52, May 2000.

[10] J. W. Haskins, Jr. and K. Skadron. Differential multithreading: Recapturing pipeline stall cycles and enhancing throughput in small-scale embedded microprocessors. In the *Workshop on Complexity-Effective Design*, Jun. 2000.

[11] 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.

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

[13] T. Mowry. Software-controlled multithreading using informing memory operations. Technical Report CMU-CS-98-169, School of Computer Science, Carnegie Mellon Univ.; www.cs.cmu.edu/tcm/tcm_papers/sw_mt_tr98.ps, Oct. 1998.

[14] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the Hydra CMP. In *Proc. of the 13th International Conference on Supercomputing*, Jun. 1999.

[15] Motorola Semiconductor Products. Motorola's DragonBall(tm) integrated processor family. www.mot.com/SPS/WIRELESS/products/DragonBall.html.

[16] T. Scherrer. Thomas Scherrer Z80-family official support page. www.geocities.com/SiliconValley/Peaks/3938/z80_home.htm.

[17] 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.

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

[19] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. www.specbench.org/osg/cpu95.

[20] 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.

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

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