Appears in *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

Mini-threads: Increasing TLP on Small-Scale SMT Processors

Joshua Redstone Susan Eggers Henry Levy
University of Washington
{redstone,eggers,levy}@cs.washington.edu

Abstract

Several manufacturers have recently announced the first simultaneous-multithreaded processors, both as single CPUs and as components of multi-CPU chips. All are small scale, comprising only two to four thread contexts. A significant impediment to the construction of larger-scale SMTs is the register file size required by a large number of contexts. This paper introduces and evaluates mini-threads, a simple extension to SMT that increases thread-level parallelism without the commensurate increase in register file size. A mini-threaded SMT CPU adds additional per-thread state to each hardware context; an application executing in a context can create mini-threads that will utilize its own per-thread state, but share the context's architectural register set. The resulting performance will depend on the benefits of additional TLP compared to the costs of executing mini-threads with fewer registers. Our results quantify these factors in detail and demonstrate that mini-threads can improve performance significantly, particularly on small-scale, space-sensitive CPU designs.

1. Introduction

Simultaneous Multithreading (SMT) is a latency-tolerant CPU architecture that adds multiple hardware contexts to an out-of-order superscalar to dramatically improve machine throughput [14, 32, 25, 13]. Recently, several manufacturers have announced small-scale SMTs (e.g., 2 to 4 thread contexts), both as single CPUs and as components of multiple CPUs on a chip [12, 30]. While these small-scale SMTs increase performance, they still leave modern wide-issue CPUs with underutilized resources, i.e., substantial performance potential is still untapped.

A primary obstacle to the construction of larger-scale SMTs is the register file. On the Alpha architecture, for example, an 8-context SMT would require 896 additional registers compared to a superscalar of similar structure. In terms of area, Burns and Guadiot [6] estimate that adding 8 SMT contexts to the R10000 would increase the register file and renaming hardware from 13% to 30% of the processor core. On Compaq's 4-context SMT, the Alpha 21464, the register file would have been *3 to 4 times* the size of the 64KB instruction cache [23]. In addition to the area requirements, the large register file either inflates cycle time or demands additional stages on today's aggressive pipelines; for example, the Alpha 21464 architecture would have required three cycles to access the register file [23]. The additional pipeline stages increase the branch misprediction penalty, increase the complexity of the forwarding logic, and compound pressure on the renaming registers (because instructions are in flight longer). Alternatively, lengthening the cycle time to avoid the extra pipeline stages directly degrades performance by reducing the rate at which instructions are processed.

This paper proposes and provides an initial evaluation of a new mechanism to boost thread-level parallelism (and consequently throughput) on small-scale SMTs, without the commensurate increase in register file size. The mechanism, called *mini-threads*, alters the basic notion of a hardware context. On the hardware level, mini-threads add additional per-thread state (aside from general purpose registers) to each SMT hardware context. Using this hardware, an application can exploit more thread-level parallelism within a context, by creating multiple mini-threads that will *share* the context's architectural register set. We denote as $_{mt}$SMT an SMT with mini-threads, and use the notation $_{mt}SMT_{i,j}$ to indicate an $_{mt}$SMT that supports i hardware contexts with j mini-threads per context. For example, an $_{mt}SMT_{4,2}$ – a 4-context $_{mt}$SMT – has the potential to deliver the same thread-level parallelism as an 8-context SMT, but with half the number of registers and greatly reduced interconnect.

Mini-threads improve on traditional SMT processors in three ways. First, mini-threads conserve registers, because each executing mini-thread does not require a full architectural register set. Second, $_{mt}$SMT allows *each* application the freedom to trade-off ILP for TLP within its hardware contexts. Applications can choose to use mini-threads to increase TLP, or to ignore them to maximize the performance of an individual thread. In the latter case, where the application dedicates its context to a single

thread, the processor performs identically to SMT. Because of this, for single-program workloads, $_{mt}$SMT will always perform better than or equal to SMT.

Third, in addition to the savings in registers, mini-threads open up new possibilities for fine-grained thread programming. Each application can choose how to manage the architectural registers among the mini-threads that share them. For example, mini-threads can simply partition the architectural register set. A wide range of more complex schemes also exist, including sharing register values among mini-threads and even dynamically distributing registers to mini-threads as their execution-time needs change. While sharing the architectural register set among mini-threads within a context creates many opportunities to optimize architectural register usage, it also introduces some interesting problems and trade-offs as well, some of which we examine here.

The principal goals of this paper are to (1) introduce the concept of mini-threads on SMT, mapping out the breadth of the design space it involves, and (2) perform an initial evaluation on one part of that design space to show that there is a potential performance gain from adopting $_{mt}$SMT. There are many possible ways in which mini-threads can be applied in both hardware and software. In this paper, we chose to evaluate the most straightforward of these: two mini-threads per context with static partitioning of the architectural register file between the mini-threads. Only if that scheme provides benefit is it worth exploring more complex schemes (and issues) that require a larger amount of effort, such as communicating and synchronizing through mini-thread-shared registers.

For all schemes, two opposing factors determine the performance of $_{mt}$SMT versus SMT. On the one hand, extra mini-threads per context may boost performance by increasing TLP, thereby increasing instruction throughput. On the other hand, performance may degrade due to additional spill code, since each mini-thread is limited to a subset of the architectural registers. $_{mt}$SMT wins when the TLP benefits of additional mini-threads outweigh the costs of fewer architectural registers per mini-thread.

This paper evaluates these opposing factors in detail using five workloads: four applications from the SPLASH-2 parallel scientific benchmark suite and the multithreaded Apache web server. These programs are naturally suited to $_{mt}$SMT because they explicitly control their degree of thread-level parallelism. In the bulk of the paper, we quantify the factors that determine $_{mt}$SMT performance; for example, we provide a detailed analysis of the changes in spill-code (which can increase as well as decrease!) due to reducing the number of registers per mini-thread. Our results show a significant improvement for $_{mt}$SMT over SMT, averaging 40% on small SMTs of 4-contexts or less, and extending even to 8-context SMTs for some applications. Perhaps surprisingly, most applications suffer only minor per-thread performance degradation when two mini-threads share a partitioned architectural register set. Thus, the increase in TLP due to the extra mini-thread translates directly into higher performance. This is particularly important for small-scale SMTs, which both need and enjoy the largest performance gains. Small SMTs are also the most practical to build, positioning $_{mt}$SMT as an important technique for realistic implementations.

The rest of this paper proceeds as follows. Section 2 defines the $_{mt}$SMT architecture, the mini-thread programming model, and their operating system and run-time support. In Section 3 we discuss the methodology of the simulator, workloads, and compilers used in our simulation-based experiments. Section 4 presents results that quantify the factors that contribute to $_{mt}$SMT performance: the benefits of adding thread-level parallelism and the costs of reducing the number of registers available to each mini-thread. In Section 5 we tie this analysis together with results that show the overall performance benefit of $_{mt}$SMT, when all factors are taken into account. Section 6 reviews previous research related to our study. We conclude in Section 7.

2. The $_{mt}$SMT architecture

$_{mt}$SMT improves on SMT by introducing architectural modifications and a thread model that optimizes architectural register usage. This section describes the basic $_{mt}$SMT architecture, its programming model, and operating system and run-time support for mini-threads.

2.1 $_{mt}$SMT architecture

An SMT processor can issue multiple instructions from multiple threads each cycle. An SMT *context* is the hardware that supports the execution of a single thread, such as a PC, a return stack, re-order and store buffers and exception handling and protection registers. Each context also contains a set of architectural registers, distinct from the architectural registers of all other contexts. A thread executing in one context cannot access the architectural registers of a thread in a different context. (A more in-depth discussion of SMT can be found in [31].)

$_{mt}$SMT augments SMT by adding to each context the hardware needed to support an additional executing thread *except for the registers*. The key feature of $_{mt}$SMT is that the mini-threads in a context *share* the context's architectural register set. Specifically, when two instructions from two different mini-threads in the same context reference

the same *architectural* register, they inherently reference the same *physical* register, accessing the same value. Note that the register renaming hardware has not changed – the mapping from architectural registers to physical registers proceeds exactly as it does on SMT. What has changed is the way that architectural register numbers are mapped to locations in the renaming table (and from there renamed to physical register numbers). Figure 1 depicts this register sharing on an $_{mt}SMT_{2,2}$. The mini-threads executing on the two PCs in each context map the same architectural register set.

$_{mt}$SMT requires hardware similar to adding extra contexts to SMT. For example, a 4-context $_{mt}$SMT with 2 mini-threads per context closely resembles an 8-context SMT in terms of the number of mini-thread hardware resources, such as re-order and store buffers and return stacks. A few additional registers are also required beyond that on a 4-context SMT to support per-mini-thread exception handling and protection (~22 registers on the Alpha 21264 [8]). However, $_{mt}$SMT has the reduced register hardware, renaming complexity and register file access time of the 4-context SMT.

Although the implementation of architectural register sharing between mini-threads on $_{mt}$SMT requires minimal hardware modification, the sharing itself creates a very different architectural interface. We begin with new terminology. Analogous to a context on SMT, a *mini-context* refers to the hardware necessary to execute a mini-thread. The architectural interface of a mini-context resembles that of an SMT context, including the architectural register set. However, on $_{mt}$SMT, all mini-contexts within the same context share this architectural register set. Therefore a mini-thread must manage the sharing of its registers in cooperation with the other mini-threads that execute within the same context. i.e., a mini-thread must be specifically compiled to execute in a mini-context.

The following section describes how mini-threads manage the sharing of their register sets.

2.2 $_{mt}$SMT programming model

On SMT, a program begins execution when the operating system first schedules it on a context, starting at the main program entry point. Later, the program may fork an additional thread by calling a *thread-fork* function and passing it the starting PC for the new thread. After the fork, the original thread continues executing in its context and the new thread begins execution on a different context. Each thread references its own distinct set of (architectural and physical) registers; threads communicate through shared memory.

On $_{mt}$SMT, a program starts as a single mini-thread at

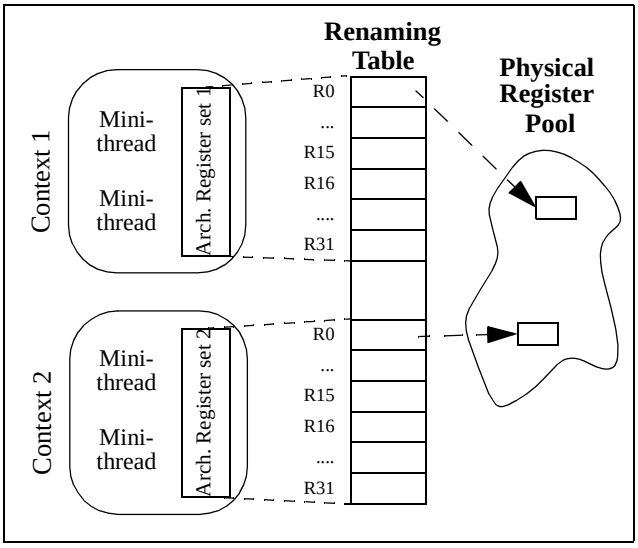


FIGURE 1. Register sharing among mini-threads on an $_{mt}SMT_{2,2}$. There are two hardware contexts, each supporting two mini-threads that share architectural registers within the context.

the main program entry point and executes in one of the mini-contexts. To create a second mini-thread, it calls a *mini-thread-fork* function. The two mini-threads share the same architectural register set, and, because they were compiled as a mini-threaded program, they have arranged *a priori* (through the compiler) to coordinate their register usage.

Mini-threads can manage their registers in a variety of ways, including statically or dynamically partitioning registers among mini-threads, sharing registers, i.e., sharing values, or a combination. Our models for register set sharing are similar to those proposed by Waldspurger and Weihl [34] for the April coarse-grain multithreaded processor [1]. $_{mt}$SMT permits all of these variations, because the application controls what register allocation is used and when. In all cases a compiler would have to compile a mini-thread for a specific scheme.

As an initial evaluation of mini-threads, this paper focuses on one of these alternatives: statically partitioning each architectural register set in half between two mini-threads. There are two ways to achieve this partitioning. In the first, each mini-thread is compiled for different architectural registers within a register set. An alternative strategy compiles both mini-threads for the *same* subset of the architectural registers and differentiates between them with an extra software-programmable state bit. The decode stage of the pipeline inserts this bit into the high-order bit of an instruction register field before accessing a register. Thus, the bit is set to 1 on the mini-context using the upper half of the register set, and to 0 on the other mini-context. With this scheme, an application compiled

to use the lower half of the register set will run correctly on either of the mini-contexts.

2.3 Operating system and run-time support

The new architectural model introduced by mini-threads creates several new operating system and run-time support issues. First, because mini-threads execute within both run-time and OS procedures, those procedures must be compiled to use registers compatibly with the executing mini-thread. Second, when entered by one mini-thread, the OS must protect its registers from modification by other mini-threads executing in the same context, since they share the same architectural register set. Third, one or both of the OS and runtime must be modified to support mini-thread management operations.

We see two application environments for mini-threads and propose a run-time and OS support solution appropriate to each. One environment is dedicated and homogeneous, e.g., a dedicated server in which all threads run instances of the same code. The OS and runtime would be compiled specifically for this environment, to allow maximum concurrency. The second environment is heterogeneous, i.e., different contexts execute different programs, some multithreaded and some not. Here the standard OS and runtime could be used, with only one mini-thread in a context allowed to execute within them at a time (an approach also used on non-symmetric multiprocessors).

The first environment is exemplified by web servers. Here, all of the processes execute identical copies of the same server code, each handling a different request. The entire system is set up (statically) for that purpose (it is not, for example, running scientific programs at the same time). In this environment a single version of the OS and runtime executes, compiled to use half of the register set. The hardware partition bit described in Section 2.2 allows a single OS/runtime image to execute on either mini-context and isolates each mini-thread's registers from the other's. When the partition bit is used, the mini-contexts are indistinguishable from distinct contexts on an SMT, albeit with fewer architectural registers. Hence, OS and runtime code that manages contexts on SMT can manage mini-threads with minimal modification. (Most of the OS can be compiled automatically to use half of the register set, except for the small number of assembly language files, which require manual register specification.)

This OS and runtime solution grants complete freedom to mini-threads to execute independently. Specifically, both mini-threads in a context may execute in the OS simultaneously, a performance-critical capability for OS-intensive workloads such as Apache (which spends 75% of its time within the OS [25]). This performance advantage comes at the price of a loss in mini-thread flexibility to manage their common architectural registers. All mini-threads must partition the register set identically and cannot share any register values. Because of their homogeneity, mini-threads in web-server-like workloads perform well despite this restricted functionality.

The second environment is typified by a multiprogrammed workload in which some programs use mini-threads and others do not. In this environment we execute a single operating system image compiled to use the *full* architectural register set, as on a standard SMT. The OS remains mostly ignorant of mini-threads, deferring mini-thread management to the runtime.

Applications that choose not to use mini-threads execute in the OS exactly as on SMT. However, when a mini-threaded application traps into the OS, the hardware blocks all other mini-threads in the same context. This guarantees that only one mini-thread per context executes in the kernel at a time, thereby protecting shared kernel registers from the actions of other mini-threads. On a normal SMT, the kernel saves the PC and registers on a trap before executing; for $_{mt}$SMT, we modify the trap handler to save the PCs, registers, and mini-thread IDs of both the *trapping* mini-thread and the *blocked* mini-threads within the context. After saving this state, the OS continues normal execution within the full register set; the mini-thread state is then restored on return to user mode. The OS remains ignorant of mini-threads, except for the extra state saved and restored on traps.

The Tru64 UNIX OS that we use already supports a form of scheduler activations [2], which allows the kernel to communicate thread activity to the pthreads runtime. A scheduler activation is a kernel upcall into the user-level threads package that alerts it about kernel scheduling events. This kernel-to-runtime communication is modified to also include information about blocked mini-threads in a context. If a trapping mini-thread blocks in the kernel, the kernel communicates the PC, registers, and minicontext ID of the other (hardware-blocked) mini-threads as part of an upcall to the user-level runtime system. The pthreads runtime views mini-threads as simply user-level threads that have scheduling constraints, and hence, it requires few modifications. In our example, when the runtime receives the kernel upcall, it can place the hardware-blocked mini-threads on a queue; when the event that caused the trapping mini-thread to block in the kernel is resolved, the kernel again performs a runtime upcall, this time identifying the state of the trapping mini-thread to the runtime, which then schedules all mini-threads in the context to run.

In a statically partitioned, two-mini-thread environment this requires two versions of the runtime, one compiled for each register usage convention (i.e., 16 and 32 registers). Each application links to its appropriate version. There is ample precedence for multiple runtimes on a single plat-

form. For example, Microsoft Windows OS supports multiple versions of library routines; 32-bit and 64-bit programs executing on 64-bit x86-compatible processors have to link to different versions. Similarly, many applications already install private versions of library routines for their own use. In SPLASH-2-like workloads all threads are compiled to use the same runtime, and therefore avoid the issue of multiple runtime copies.

In summary, we take two approaches. For the server approach, which is OS-intensive, we recompile the OS and runtime to allow mini-threads in a context to execute them simultaneously, at the cost of limiting register-sharing flexibility among mini-threads. For the multiprogramming approach, we place no restrictions on register usage and sharing among mini-threads, but allow only one mini-thread within the OS at a time.

3. Simulation and evaluation infrastructure

In this section we describe the infrastructure and methodology used in our simulation-based experiments.

3.1 SMT and superscalar simulation methodology

Our simulator is an enhanced version of the Alpha-based SMT simulator used in [25]. It combines an execution-driven processor simulator with the SimOS [26] machine simulation framework. The processor simulator models the pipeline and the memory system in great detail. (Table 1 lists the base processor parameters; see [23] for a more complete list). The SMT pipeline consists of 9 stages, with 2 stages each dedicated to reading and writing the large register file. The superscalar simulations use a shorter 7-stage pipeline, since they lack the need for additional read and write stages. In this work all processors are configured identically except for the number of contexts and pipeline stages.

The SimOS framework allows us to include in our simulation all operating system activity, including PAL code, interrupts, and network activity. It models the privileged processor state and the non-processor components of a machine in enough detail to boot and execute a version of Compaq Tru64 Unix adapted to SMT.

We emulate $_{mt}$SMT by compiling applications to use fewer registers and simulating the applications on a standard SMT. For example, to model an $_{mt}SMT_{4,2}$ we compile an application into threads that use only 1/2 the normal registers and simulate it on an 8-context SMT. This methodological simplification does not affect performance; each context touches no more registers than would be available on $_{mt}$SMT. We choose this methodology because it greatly simplifies compilation and allows us flexibility in choosing which specific registers to use. Section 3.3 discusses compiler support in more detail.

TABLE 1. SMT parameters.

Fetch Policy	8 instructions per cycle from up to 2 contexts (the 2.8 ICOUNT scheme of [31])
Functional Units	6 integer (including 4 Load/Store and 1 Synchronization unit); 4 floating point
Instruction Queues	32-entry integer and floating point queues
Renaming Registers	100 integer and 100 floating point
Retirement bandwidth	12 instructions/cycle
TLB	128-entry ITLB and DTLB
Branch Predictor	McFarling-style, hybrid predictor [16]
Icache	128KB, 2-way set associative, single port 2 cycle fill penalty
Dcache	128KB, 2-way set associative, dual ported 2 cycle fill penalty
L2 cache	16MB, direct mapped, 20 cycle latency, fully pipelined (1 access per cycle)
L1-L2 bus	256 bits wide, 2 cycle latency
Memory bus	128 bits wide, 4 cycle latency
Physical Memory	128MB, 90 cycle latency, fully pipelined

3.2 Workloads

We evaluate $_{mt}$SMT on five programs: the Apache web server [3] and four applications from the SPLASH-2 benchmark suite [29]. All are naturally suited to SMT, because they are parallel applications that control their degree of parallelism. Apache adapts to more contexts by processing more requests in parallel, and SPLASH-2 forks multiple threads via a command-line argument.

Apache is a popular, public-domain Web server run by the majority of Web sites [19]. We generate requests with SPECWeb96, a web server performance benchmark [28]. We configure Apache with 64 server processes and SPECWeb with 128 clients that provide requests. To match the speed of the simulated server, we execute three synchronized copies of SimOS on a single Alpha: one executing Apache and two running copies of SPECWeb with 64 clients each. [25] describes this setup in more detail.

SPLASH-2 is a suite of explicitly parallel scientific applications. We evaluate four programs from this suite: Barnes, Fmm, Raytrace, and Water-spatial. We adapt each application to SMT by replacing the heavyweight synchronization primitives with the faster SMT hardware lock-based synchronization primitives [33].

Traditional throughput metrics such as IPC may not accurately reflect overall speedup for threaded programs. Instead, we use a higher-level performance metric of *work per unit time* as the basis for comparison. To define equivalent units of work, we modified each application to insert special markers at appropriate points in the code to indicate the progress that a particular thread has made. Our metric counts markers per unit time, where a marker corresponds to a unit of work.

3.3 Compiler methodology

In this work we compile applications to use only 1/2 of the architectural register set for each $_{mt}$SMT configuration. We employed a combination of Gcc 2.95.3, a modern, widely-used, open-source compiler, and Compaq C 6.4, the closed-source compiler for Tru64 UNIX systems. Gcc provides a simple command-line option to control which architectural registers are available to the register allocator when compiling a program. We compiled the SPLASH-2 applications with Gcc.

Apache requires special compiler methodology. While the SPLASH-2 applications spend a negligible amount of time in the kernel (less than 1%), Apache spends a full 75% of execution cycles in the operating system. Because OS behavior dominates the performance of this workload, any evaluation of Apache must include it.

We compiled the OS for $_{mt}$SMT with Compaq's C compiler (Gcc is unable to compile it). The Compaq C compiler lacks Gcc's simple command-line option to control register allocation. Instead, it supports a set of pragmas that manipulate register usage, and require manual modifications to each OS source file. For this study, we modified enough files to cover 57% of all dynamic OS instructions when executing Apache. To compensate for the unmodified portions of the kernel, we extrapolated the trends observed over the modified regions of the OS to the entire OS. We applied the same methodology to factor out the twelve percent of Apache's dynamic instructions that are from user-level shared libraries (we did not recompile shared libraries).

Modeling $_{mt}$SMT on SMT as described in Section 3.1 lets multiple mini-threads execute in the OS simultaneously. This corresponds to the OS-intensive environment described in Section 2.3. While this environment suits Apache, the SPLASH-2 applications would most likely execute in the environment in which other mini-threads in a context block when one traps. We do not explicitly model this blocking, but take it into account arithmetically. Our calculations suggest that the impact of blocking on our results would be only 1%, because SPLASH-2 spends so little time in the kernel.

4. Evaluating the register / mini-thread trade-off

$_{mt}$SMT allows applications to make a trade-off between one thread per context with a full architectural register set, or multiple mini-threads per context, each with a subset of the architectural register set. Two factors determine the efficacy of this trade-off: (1) the performance *benefit* due to the extra executing mini-threads, and (2) the performance *degradation* due to fewer available registers per mini-thread. For each of these factors, there are also two levels of potential impact. At the hardware level, each can improve or degrade IPC, and at the software level, each can alter the number of instructions per unit of work. Of these four factors, two are straightforward. The extra mini-threads allow the processor to take advantage of greater TLP, possibly increasing throughput, while the reduction in the number of registers per mini-thread burdens the register allocator, possibly resulting in increased spill code. The other two factors reflect less obvious effects. First, the extra spill code can impact cache performance, affecting IPC. Second, an increase in the number of mini-threads can increase total thread overhead, altering the number of instructions executed.

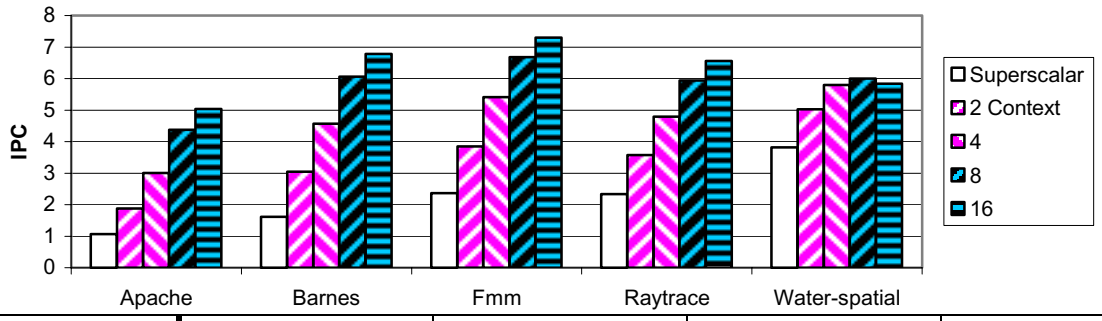
These four factors completely describe $_{mt}$SMT performance relative to a traditional SMT processor. In the interests of space, the following sections investigate only two of these, the IPC benefit of increased TLP and the increased instruction count due to fewer registers per mini-thread. Section 5 then presents overall $_{mt}$SMT performance, and quantifies the relative impact on performance of all four factors.

4.1 Throughput improvement due to the extra mini-threads

This section measures the maximum throughput improvement due solely to the addition of extra mini-threads, i.e., due to the increase in thread-level parallelism. We can compute this effect by measuring the boost in throughput that is provided by a conventional SMT with the number of hardware contexts equal to the total number of mini-threads. For example, to calculate the maximum TLP benefit of $_{mt}SMT_{2,2}$ over the base 2-context SMT without mini-threads, we compare a 4-context SMT with a 2-context SMT. Comparing throughput on these two SMT machines isolates the throughput improvement from any effects of the reduced number of registers.

The top of Figure 2 graphs SMT throughput for all SMT sizes corresponding to the $_{mt}$SMT configurations that we evaluate, ranging from a superscalar up through a 16-mini-thread machine. The table at the bottom of Figure 2 shows, for each $_{mt}$SMT, the percentage improvement in IPC over its base SMT due to the extra mini-threads. Each entry in the table represents a rough upper bound on the potential performance improvement of $_{mt}$SMT.

In most cases, doubling the number of mini-threads increases instruction throughput, and the benefit diminishes as the number of contexts increases. This is reflected by the levelling off of IPC in the graph and by the decreasing percentage IPC improvement with increasing base SMT size in the table. For example, on a 2-context SMT, the boost in IPC due to doubling the number of contexts



	$_{mt}SMT_{1,2}$	$_{mt}SMT_{2,2}$	$_{mt}SMT_{4,2}$	$_{mt}SMT_{8,2}$
Apache	76	60	45	15
Barnes	89	50	33	12
Fmm	62	41	23	9
Raytrace	53	34	24	10
Water-spatial	32	15	3	-3

FIGURE 2. Improvement in throughput due to extra contexts. The graph shows the IPC of a range of SMT sizes. The table lists the component of $_{mt}$SMT IPC performance due solely to the extra mini-threads. For example, the $_{mt}SMT_{2,2}$ column shows the percentage IPC improvement of a 4-context SMT over a 2-context SMT.

averages 40% over all workloads (the maximum is 60%), while on an 8-context SMT, the benefit averages only 9%. In other words, extra contexts are most valuable for small SMTs, which have difficulty providing enough sources of instructions to fully utilize the machine; but because the processor's execution resources are finite, the marginal benefit of additional contexts decreases as the number of contexts increases. Nonetheless, with the exception of water-spatial, adding mini-contexts provides additional instruction throughput to any base SMT.

Apache and Water-spatial embody two extremes of sensitivity to the additional TLP provided by mini-threads. Apache benefits the most from extra contexts. Its poor superscalar performance deserves most of the blame [25]. At the opposite extreme, Water-spatial squanders extra contexts. Part of the reason that Water-spatial does not successfully leverage increased TLP is its relatively high superscalar IPC. In general, the more a single thread can utilize a processor, the less opportunity exists for other instruction sources (i.e., other threads) to further increase utilization. Two other factors also limit Water-spatial's performance, and, in fact, cause IPC to drop as the number of contexts increases: the D-cache miss rate balloons from 0.3% on a 2-context SMT to 20% with 16 contexts, and the average percentage of cycles a context is blocked on a user-level lock rises from 17% to 25%.

4.2 Extra instructions due to fewer registers

This section measures a second factor affecting the register/mini-thread trade-off: the change in the number of instructions per unit of work as the number of available registers decreases. To quantify the change, we compared each workload executing on two machines: an $_{mt}$SMT, and a conventional SMT with the same number of contexts as the $_{mt}$SMT has mini-contexts. The two machines differ only in the number of architectural registers available to each thread; comparing them therefore isolates the effect of reducing the number of registers per mini-thread.

Figure 3 graphs the percentage change in dynamic instructions due to fewer architectural registers for each $_{mt}$SMT configuration. With the exception of Fmm, the applications are remarkably insensitive to the number of available registers. The increase in dynamic instructions ranged between 16% for Fmm to -7% for Barnes, with an average of 3%.

For one application, Barnes, the amount of spill code *decreases* as registers become scarce, dropping an average of 7%. The entire reduction occurred in one procedure in which the register allocator substituted caller-saved registers for callee-saved registers when the number of architectural registers was reduced. Consequently, mandatory spills at procedure entry and exit were replaced with a smaller increase in spill code in the interior of the calling procedure.

The results for Apache provide our first glimpse into the sensitivity of the operating system to the number of available registers. In fact, the graph slightly overstates kernel sensitivity, because it combines user and kernel instruction counts, and Apache's user-level instruction count rise is relatively larger (4%). Factoring out the user-level behavior leaves kernel instruction counts that barely budge upwards 0.8% as the number of architectural regis-

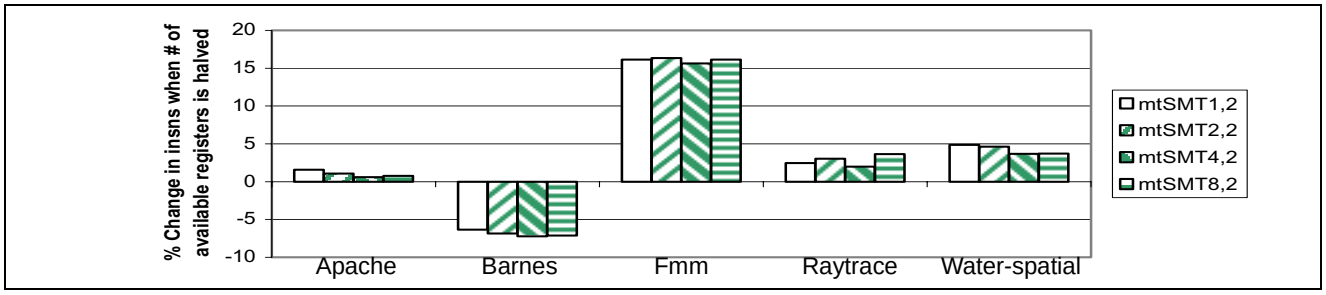


FIGURE 3. Change in instruction counts due to fewer registers per thread on $_{mt}$SMT. Each bar measures the percentage change in instruction counts between an $_{mt}$SMT configuration and an SMT which has the same number of contexts as the total number of mini-contexts in the $_{mt}$SMT.

ters decrease! Two factors contribute to the kernel's insensitivity to a reduced register set. First, the frequency of pointer usage in the kernel prevents the register allocator from keeping many values in registers. Second, simple operations with short-lived values, such as checking permissions or error conditions, dominate OS activity, leading to a low average number of simultaneously live values.

For most programs, loads and stores to the stack, primarily for procedure call handling, constitute the bulk of spill instructions that are generated with the 32-register compile. As the number of available registers decreases, the rise in stack operations causes the total number of loads and stores to increase slightly from an average of 32% to 37% of all instructions. In addition, spilling within a procedure rather than around procedure calls begins to dominate, with the largest increase coming from register moves. Two effects contribute to the increase in non-load-store spill code as the number of available registers decreases. First and foremost, the compiler generates more register-to-register moves to shuffle values within the restricted set of architectural registers. Second, the register allocator chooses to undo simple CSE optimizations and recompute some constant values rather than spill them to memory, thereby generating extra non-load-store operations. [24] breaks down spill code in more detail.

4.3 Summary

Overall, adding mini-contexts increases TLP, significantly boosting IPC in most, but not all configurations. We observe increases in IPC solely due to this effect ranging from 89% down to -3%. On the other hand, reducing the number of available registers per mini-thread generally degrades performance due to increased spill code. The extra spill code both increases the dynamic instruction count and negatively impacts IPC by raising the number of Dcache and DTLB misses (data not shown).

5. Performance on $_{mt}$SMT

The previous section examined in isolation two of the four factors that contribute to $_{mt}$SMT performance. This section addresses two remaining questions. First, how do the four factors relate in importance to each other? And second, does $_{mt}$SMT improve overall performance?

To enable an intuitive comparison of the different factors, in this section we present a stacked bar chart (Figure 4) that combines the impact of each factor on $_{mt}$SMT's performance relative to SMT. We cannot graph the factors directly, because they are related multiplicatively (see [24] for the derivation), and a stacked bar chart expresses additive relationships. To convert the relationship of the factors to an additive one, we take the logarithm of each factor, representing each log term as a bar segment. The advantage of graphing the logarithm of each factor is that the relationship between bar heights becomes intuitive. Two factors of equal magnitude will be the same height, and if they have different signs, they will cancel each other's effect on performance.

The triangles in Figure 4 signify the overall performance improvement of $_{mt}$SMT over SMT, taking into account the factors' positive and negative effects. The magnitude of a bar segment measures the contribution to speedup of a particular factor. Note that the height of any bar or combination of bars can only be interpreted against the y-axis if it is transposed to the origin. Table 2 echoes the triangles in the graph, reporting the total percentage $_{mt}$SMT speedup.

Apache benefits the most from mini-threads and these benefits appear on all $_{mt}$SMT configurations. However, the greatest performance improvement occurs on the smallest SMTs, where TLP is the most limited. For example, adding an extra mini-thread to a superscalar (the bar at the left of the graph) increases Apache's request throughput 83%. As the size of the SMT grows, the performance improvement due to mini-threads progressively decreases. However, even on an 8-context SMT, trading off registers

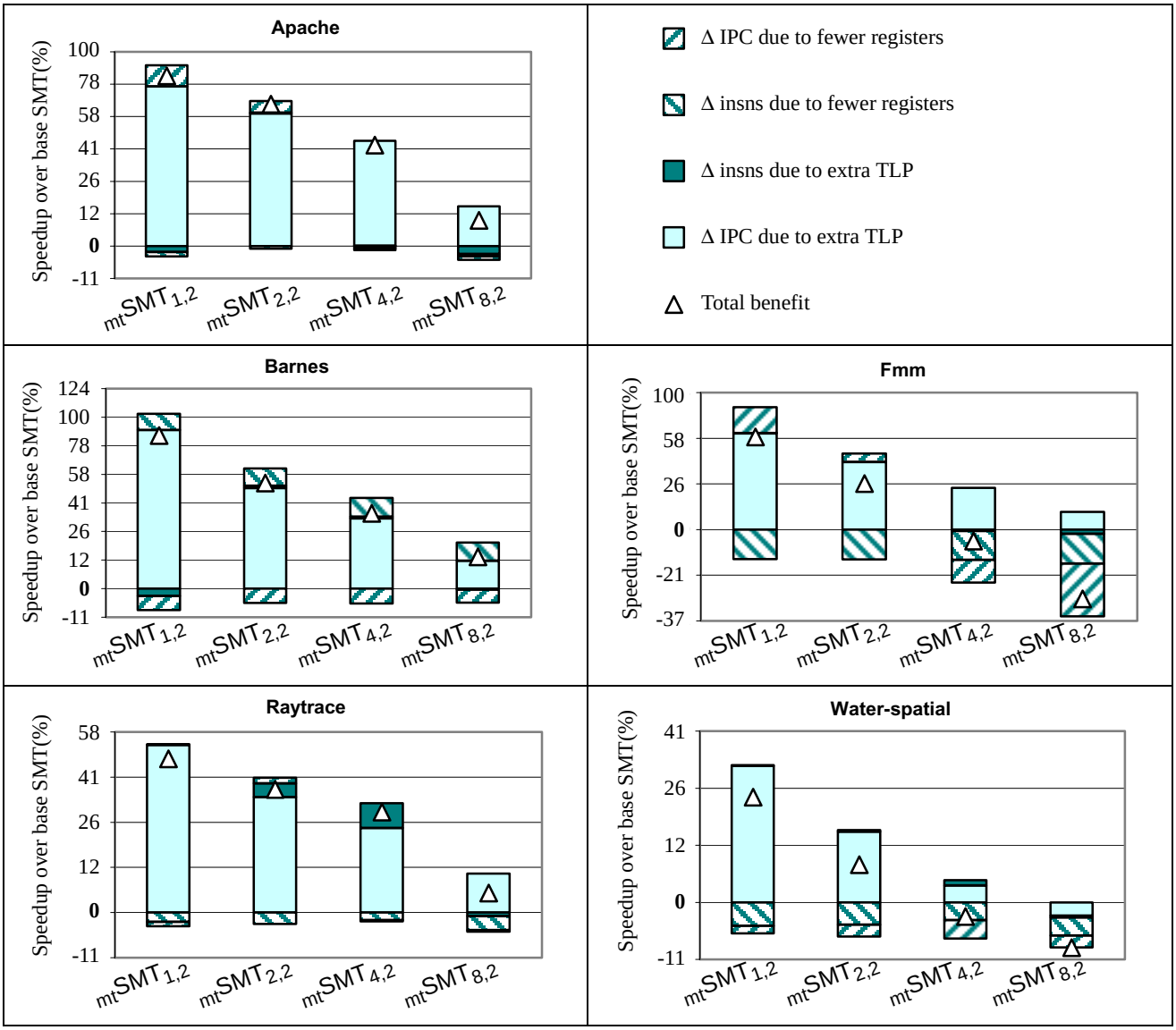


FIGURE 4. Performance improvement of $_{mt}$SMT over SMT, broken down by factor. The x-axis enumerates different $_{mt}$SMT configurations, arranged in order of increasing total number of mini-contexts. Each column consists of four bars and measures the speedup of $_{mt}$SMT compared to the corresponding base SMT. Each of the four segments of each bar measures the change in performance due to that factor. The y-axis indicates percentage speedup. Note that the y-axis scale differs for each application. The triangle in each column represents the sum of the heights of the bars in the column which equals the total $_{mt}$SMT speedup.

for contexts boosts request throughput by 10%[1].

The SPLASH-2 applications also benefit from mini-threads, although on average less than Apache. Mini-threads improve the performance of all applications on the smaller SMTs. For example, speedups on a 2-context SMT lie between 8% to 53%, with a median improvement of 32%. For half of the applications, the improvement also scales to larger SMTs. Barnes and Raytrace see speedups averaging 32% with 4 contexts and 9% with 8. The other two applications do not benefit on either of the larger SMTs, with average performance degrading by 4% and 20% on the 4- and 8-context SMTs, respectively.

The magnitude of each factor explains why performance improves so much. For most applications and most $_{mt}$SMT configurations, the IPC boost due to extra mini-threads far dominates any other factor. With the exception of Fmm and Water-spatial executing on larger SMTs, the

1. At 16 contexts, hardware context 0 becomes a performance bottleneck, because certain OS activities such as network interrupts are funneled through it, resulting in 20% idle time on other contexts.

TABLE 2. Total percentage $_{mt}$SMT speedup

	$_{mt}SMT_{1,2}$	$_{mt}SMT_{2,2}$	$_{mt}SMT_{4,2}$	$_{mt}SMT_{8,2}$
Apache	83	66	43	10
Barnes	85	53	36	14
Fmm	60	26	-6	-30
Raytrace	48	37	29	5
Water-spatial	24	8	-3	-9

IPC benefit of the extra mini-threads averages 41%. Reducing the number of available registers has little effect on performance, allowing the IPC boost to translate directly into improved overall $_{mt}$SMT performance. The performance decline from decreasing the number of registers available to each mini-thread (effect on IPC combined with executing more instructions) averaged only 2%.

For Fmm and Water-spatial, the large cost of reducing the number of registers and the relatively small boost in IPC due to extra mini-threads meant that $_{mt}$SMT did not pay off for larger SMT configurations. The overall impact of reducing the number of registers averaged -29% for Fmm and -6% for Water-spatial. In contrast, the overall benefit of the extra mini-threads averaged only 15% and 1%, respectively, insufficient to overcome the cost of the reduced number of registers.

Surprisingly, the IPC impact of the increase in instructions equaled or exceeded that of the instructions themselves. In 35% of all configurations and applications examined, the IPC impact of fewer registers exceeded the percentage increase in the dynamic instruction count. In 70% of all configurations, the IPC impact was at least 50% of that of the instruction count.

Overall, our results show that trading off registers for contexts improves performance, especially on the smaller SMTs that are characteristic of the first commercial implementations. In particular, performance improved by an average of 38% (averaged over all applications) on a 2-context SMT, with decreasing improvements on successively larger SMTs. In the above experiments, we forced applications to use mini-threads. If we allow them instead to use mini-threads only when advantageous (as they can do, since employing mini-threads is an application-specific decision), then the average performance improvement on 4- and 8-context SMTs is 22% and 6%, rather than 20% and -2%, respectively.

Finally, to further map out the mini-thread design space, we evaluated SPLASH-2 applications on $_{mt}$SMTs with three mini-threads per context, compiling applications to use one third of the register set (with a few registers left over). On a two-context $_{mt}$SMT, three mini-threads raised the average performance improvement compared to SMT to 43% from 31% with two mini-threads. On larger SMTs, they performed worse than two mini-thread $_{mt}$SMTs, because larger SMTs benefit less from extra mini-threads and the even further reduced number of registers induced more spill code.

6. Related work

Many researchers have explored methods for conserving registers in various contexts. Closest to our work is Waldspurger and Weihl's study of register relocation [34] in the April/Alewife processor, a distributed shared-memory multiprocessor that uses software multithreading to tolerate latencies from remote memory references and failed synchronization attempts [1]. April executes one thread at a time until it incurs a miss, at which point a hardware trap signals the OS to context switch to another thread. Waldspurger and Weihl propose treating the 4 hardware contexts of the April CPU as a single large register file, partitioning the entire file in software. Their scheme uses a register relocation mask mechanism to offset thread-local register numbers into the large register file. The compiler must ensure protection between all of the threads executing on the machine; hence, all loaded threads are assumed to be part of a single application. Waldspurger and Weihl evaluate this scheme for April using a synthetic workload with stochastic run lengths and varying inter-fault latencies to show how utilization changes with inter-fault latency for alternative register management schemes.

In contrast, our work exists in a more realistic and modern hardware and software environment. We evaluate intra-context architectural register partitioning among mini-threads for an out-of-order, simultaneously-multithreaded CPU with multiple hardware contexts and register renaming. We simulate real parallel programs and a multithreaded server compiled for mini-threads, include operating system code, and we provide detailed measurement and analysis of all of the factors that influence performance.

Several researchers have investigated adding special-purpose PCs with reduced register requirements to a superscalar, mostly for the purpose of improving performance of a primary thread by prefetching or warming up the branch prediction hardware [11, 4, 35, 27, 7]. These special purpose PCs usually lack an independent set of registers, and instead share registers with the primary register set and/or have private registers written by hardware. The special PCs begin execution on a cache miss or, alternatively, by request from the primary thread [35].

Mowry and Ramkissoon investigate software-based multithreading on an architecture in which a cache miss causes the processor to branch to a predetermined user-level PC [18]. They set this PC to a light-weight context-switch routine and suggest compiler-based register-file

partitioning to reduce context-switch overhead.

Multiple threads executing within a single stack frame and possibly a single register set have been investigated in the context of dataflow architectures on large parallel machines, such as *T, pRISC, and TAM [20, 21, 10]. In these architectures, threads consists of only a few instructions, and are used to hide latencies of, for example, memory operations. The compiler manages register and stack frame usage between threads. This management is typically conservative because of the large number of threads and the uncertainty in their dynamic execution order. For example, threads may not rely on any values in registers when they begin execution.

Every architecture designer must decide how many registers to support in the ISA. Researchers have investigated the sensitivity of applications to the number of architectural registers. Bradlee et al. [5] found that reducing the number of integer registers from 32 to 16 had a negligible effect on execution time on some integer programs, and caused a 17% degradation on scientific applications. Postiff et al. [22] argue that application sensitivity to the number of architectural registers increases as compiler technology improves.

$_{mt}$SMT conserves registers by mapping a single architectural register set among multiple mini-threads. Researchers have explored other ways to reduce the register file burden in architectures. Cruz et al. [9] suggest structuring the register file as a multi-level cache, complete with a pseudo-LRU replacement policy. They find that, due to the savings in access time, such an organization outperforms a non-pipelined, single banked architecture by 90% for the SPEC95 benchmarks. Monreal et al. [17] focus on conserving renaming registers by delaying the pipeline stage at which physical registers for destination operands are allocated. They find a 25% reduction in the number of renaming registers with little loss in performance. Lo et al. [15] investigate deallocating registers on SMT after their last use via compiler-inserted annotations. They observed up to an average speedup of 60% with the most efficient annotation mechanisms. They also found that deallocating the registers of idle contexts supports a 25% reduction in the number of registers on a 4-context SMT with no loss in performance. Lo and Monreal both focus on improving the sharing of renaming registers. $_{mt}$SMT focuses on economizing architectural registers as well as renaming registers, and could work synergistically with both of their techniques. All three of these techniques could benefit from Cruz's optimizations.

7. Conclusion

Small-scale SMTs will clearly become a part of the CPU landscape in the next several years. While such CPUs can obtain significant performance improvements through multithreading, they are still likely to underutilize the massive processing and storage resources available on the next generation of out-of-order processors.

This paper has introduced and evaluated a simple modification to SMT that greatly increases throughput. This modification, called mini-threads, adds partial thread-state hardware to each context, allowing mini-threads executing within the same context to *share* its architectural register set. Consequently, mini-threads allow processor implementers to increase the degree of parallelism supported by SMT by avoiding a primary obstacle to scaling up SMT: the size of the register file and renaming hardware. Essentially, they propel SMT further along the throughput curve.

Implementing mini-threads allows applications to trade off register set size for TLP. Each application decides independently whether or not to use mini-threads. If it ignores the mini-contexts, the machine behaves identically to an SMT. Alternatively, if it chooses to create mini-threads, it can boost TLP and machine throughput. However, since mini-threads in each context share the architectural register set, the application must arrange for its threads to manage it. Because of the flexibility of applications to use mini-threads only when beneficial, adding mini-thread contexts to SMT will never degrade performance for single-program workloads.

This paper evaluates mini-threads with a statically partitioned register model and two mini-threads per context on the Apache web server and 4 SPLASH-2 parallel scientific applications. Overall, our results show that trading off register set size for TLP improves performance, especially on the smaller SMTs, which are characteristic of the first commercial implementations. (Adding mini-threads improves performance over all applications evaluated by an average of 38% (and a maximum of 66%) on a 2-context SMT, with decreasing improvements on larger SMTs.) The reason for the large improvement is that almost all applications can exploit the extra mini-threads to boost IPC, and most suffer only minor performance degradations due to partitioning the register set. In particular, restricting applications to half of the register set degraded performance by only 5% on average. However, the TLP benefit due to the extra mini-threads ranged from 41% on a small 2-context SMT to 7% on an 8-context SMT.

While in this work we partitioned the register set equally among the mini-threads that share it, nothing in the mini-thread architecture precludes other schemes, should appropriate compiler and programming technology be developed. Mini-threads also allow a variable partitioning of the register file adapted to the needs of particular mini-threads and the sharing of register values between mini-threads. We plan to explore these more aggressive schemes as future work.

8. Acknowledgements

This research was supported by NSF grants ITR-0085670, CCR-0121341, CCR00-85670 and EIA96-32977. We'd like to thank Larry Ruzzo for help with the statistical analysis and the reviewers for their insightful comments.

9. References

[1] AGARWAL, A., LIM, B.-H., KRANZ, D., AND KUBIATOWICZ, J. April: A processor architecture for multiprocessing. In *Proceedings of the International Symposium on Computer Architecture* (June 1990).

[2] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Symposium on Operating Systems Principles* (October 1991).

[3] APACHE SOFTWARE FOUNDATION. *Apache Web Server.* http://www.apache.org/.

[4] BALASUBRAMONIAN, R., DWARKADAS, S., AND ALBONESI, D. H. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the International Symposium on Computer Architecture* (June 2001).

[5] BRADLEE, D., EGGERS, S., AND HENRY, R. The effect on RISC performance of register set size and structure versus code generation strategy. In *Proceedings of the International Symposium on Computer Architecture* (May 1991).

[6] BURNS, J., AND GAUDIOT, J.-L. Quantifying the SMT layout overhead-does SMT pull its weight? In *International Symposium on High-Performance Computer Architecture* (January 2000).

[7] CHAPPELL, R., STARK, J., KIM, S., REINHARDT, S., AND PATT, Y. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the International Symposium on Computer Architecture* (May 1999).

[8] COMPAQ. *Alpha 21264 Hardware Reference Manual*. http://www.support.compaq.com/alpha-tools/documentation/current/chip-docs.html.

[9] CRUZ, J.-L., GONZÁLEZ, A., VALERO, M., AND TOPHAM, N. P. Multiple-banked register file architectures. In *Proceedings of the International Symposium on Computer Architecture* (June 2000).

[10] CULLER, D. E., SAH, A., SCHAUSER, K. E., VON EICKEN, T., AND WAWRZYNEK, J. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991).

[11] GWENLAPP, L. Dansoft develops VLIW design. *Microprocessor Report 11*, 2 (February 1997).

[12] INTEL. *Hyper-Threading Technology.* http://developer.intel.com/technology/hyperthread/.

[13] LO, J., BARROSO, L., EGGERS, S., GHARACHORLOO, K., LEVY, J., AND PAREKH, S. An analysis of database workload performance on simulataneous multithreading processors. In *Proceedings of the International Symposium on Computer Architecture* (June 1998).

[14] LO, J., EGGERS, S., EMER, J., LEVY, H., STAMM, R., AND TULLSEN, D. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems 15*, 2 (August 1997).

[15] LO, J., PAREKH, S., EGGERS, S., LEVY, H., AND TULLSEN, D. Software-directed register deallocation for simultaneous multithreading processors. *IEEE Transactions on Parallel and Distributed Systems* (September 1999).

[16] MCFARLING, S. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab (June 1993).

[17] MONREAL, T., GONZÁLEZ, A., VALERO, M., GONZÁLEZ, J., AND NALS, V. V. Delaying physical register allocation through virtual-physical registers. In *International Symposium on Microarchitecture* (November 1999).

[18] MOWRY, T. C., AND RAMKISSOON, S. R. Software-controlled multithreading using informing memory operations. In *International Symposium on High-Performance Computer Architecture* (January 2000).

[19] NETCRAFT. *Netcraft Web Server Survey.* http://www.netcraft.com/survey/.

[20] NIKHIL, R. S. Can dataflow subsume von neumann computing? In *Proceedings of the 16th annual international symposium on Computer architecture* (June 1989).

[21] NIKHIL, R. S., PAPADOPOULOS, G. M., AND ARVIND. *T: A multithreaded massively parallel architecture. In *Proceedings of the International Symposium on Computer Architecture* (May 1992).

[22] POSTIFF, M., GREENE, D., AND MUDGE, T. The need for large register files in integer codes. Technical Report CSE-TR-434-00, EECS/CSE University of Michigan (July 2000).

[23] PRESTON, R., BADEAU, R., BAILEY, D., BELL, S., BIRO, L., BOWHILL, W., DEVER, D., FELIX, S., GAMMACK, R., GERMINI, V., GOWAN, M., GRONOWSKI, P., JACKSON, D., MEHTA, S., MORTON, S., PICKHOLTZ, J., REILLY, M., AND SMITH, M. An 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers* (February 2002).

[24] REDSTONE, J. *An Analysis of Several Software Interface Issues for SMT Processors*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA (2002).

[25] REDSTONE, J., EGGERS, S. J., AND LEVY, H. M. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (November 2000).

[26] ROSENBLUM, M., BUGNION, E., HERROD, S., WITCHEL, E., AND GUPTA, A. The impact of architectural trends on operating system performance. In *Symposium on Operating Systems Principles* (December 1995).

[27] SONG, Y., AND DUBOIS, M. Assisted execution. Technical Report CENG 98-25, Department of EE-Systems, University of Southern California (October 1998).

[28] SPECBENCH. *An explanation of the SPECWeb96 benchmark.*

[29] STANFORD UNIVERSITY. *Stanford Parallel Applications for Shared Memory (SPLASH)*. http://www-flash.stanford.edu/apps/SPLASH/.

[30] SUN. *Sun says UltraSparc V two chips in one*. http://news.com.com/2100-1001-271135.html?legacy=cnet.

[31] TULLSEN, D., EGGERS, S., EMER, J., LEVY, H., LO, J., AND STAMM, R. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the International Symposium on Computer Architecture* (May 1996).

[32] TULLSEN, D. M., EGGERS, S., AND LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture* (June 1995).

[33] TULLSEN, D. M., LO, J. L., EGGERS, S. J., AND LEVY, H. M. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *International Symposium on High-Performance Computer Architecture* (January 1999).

[34] WALDSPURGER, C. A., AND WEIHL, W. E. Register relocation: flexible contexts for multithreading. In *Proceedings of the International Symposium on Computer Architecture* (May 1993).

[35] ZILLES, C., AND SOHI, G. Execution-based prediction using speculative slices. In *Proceedings of the International Symposium on Computer Architecture* (July 2001).