Improving Server Software Support for Simultaneous Multithreaded Processors

Luke K. McDowell, Susan J. Eggers and Steven D. Gribble

University of Washington
Department of Computer Science and Engineering
Box 352350
Seattle, WA 98195

{lucasm, eggers, gribble}@cs.washington.edu

ABSTRACT

Simultaneous multithreading (SMT) represents a fundamental shift in processor capability. SMT's ability to execute multiple threads simultaneously within a single CPU offers tremendous potential performance benefits. However, the structure and behavior of software affects the extent to which this potential can be achieved. Consequently, just like the earlier arrival of multiprocessors, the advent of SMT processors prompts a needed re-evaluation of software that will run on them. This evaluation is complicated, since SMT adopts architectural features and operating costs of both its predecessors (uniprocessors and multiprocessors). The crucial task for researchers is to determine which software structures and policies – multiprocessor, uniprocessor, or neither – are most appropriate for SMT.

This paper evaluates how SMT's changes to the underlying hardware affects server software, and in particular, SMT's effects on memory allocation and synchronization. Using detailed simulation of an SMT server implemented in three different thread models, we find that the default policies often provided with multiprocessor operating systems produce unacceptably low performance. For each area that we examine, we identify better policies that combine techniques from both uniprocessors and multiprocessors. We also uncover a vital aspect of multi-threaded synchronization (interaction with operating system thread scheduling) that previous research on SMT synchronization had overlooked. Overall, our results demonstrate how a few simple changes to applications' run-time support libraries can dramatically boost the performance of multi-threaded servers on SMT, without requiring modifications to the applications themselves.

Categories and Subject Descriptors

C.1.4[Processor Architectures]: Parallel Architectures -- SMT

D.4.1[Operating Systems]: Process Management -- multiprocessing, synchronization, threads.

D.4.2[Operating Systems]: Storage Management -- allocation/deallocation strategies.

General Terms

Performance

Keywords

Simultaneous multithreading, servers, runtime support.

This work was supported by an NSF Graduate Research Fellowship for Luke McDowell, an IBM Faculty Partnership Award for Susan Eggers, and NSF grants ITR-0085670 and EIA-9632977. We'd also like thank the anonymous reviewers for their helpful comments on the paper

1 INTRODUCTION

The advent of multiprocessors drove new paradigms for software architectures. In order to take advantage of potentially linear program speedups through parallel execution, programmers decomposed their single processor applications into multiple processes, and used synchronization primitives to control access to shared, writable data. However, SMP's parallel hardware provided more than the opportunity and challenge of executing duplicated processes concurrently. Because of the fundamental change to the underlying hardware, what was previously cheap on a uniprocessor (for example, repeated accesses to cache-resident data) often became expensive on a shared memory MP (interprocessor communication over a processor-shared bus), and vice versa. To realize the speedups promised by the hardware, the software had to change in fairly fundamental ways. Consequently, over the past few decades, researchers have developed algorithms and data structures to enable software to adapt to multiprocessor hardware configurations, and also to take advantage of and avoid the pitfalls of the differences between multiprocessors and uniprocessors.

Simultaneous multithreading (SMT) [43] represents another such shift in the underlying hardware, and therefore prompts a similar exploration into software design and support. Here, however, the situation is more complicated, since SMT adopts some architectural features and operating costs of both of its predecessors. Like a multiprocessor, SMT executes multiple threads of control, and instructions from these threads execute in parallel; however, SMT's thread-shared data structures are more effectively utilized if they are organized so as to encourage sharing (even false sharing!), as is often the default organization on uniprocessors [31]. The crucial task for researchers is to determine which software data structures and policies, multiprocessor, uniprocessor, or neither, are most appropriate for SMT.

To address this challenge, this paper investigates the performance impact and optimization of three software issues that SMT hardware affects: dynamic memory allocation, stack allocation and thread synchronization. For all three, we consider specific design issues that may be trouble spots for threaded applications executing on SMT processors. First, how should dynamic memory allocation be implemented? Are SMP-like allocators appropriate, or do the unique features of SMT change the story? Second, how do SMT's thread-shared caches impact the design of static memory allocation, and in particular, thread stack allocation? Third, how should threads or processes synchronize on an SMT? Should threads block or spin? Should the algorithms involve the operating system, and to what extent are specialized hardware structures necessary for good performance? Once we have answered the above questions, we then consider the issue of where the resultant SMT-aware policies should be implemented: in the application, in applications' run-time libraries (e.g., libC or pthreads), or in the operating system.

For each of the three design issues (dynamic memory allocation, static memory allocation, and thread synchronization), we explain how SMT is different from uniprocessors and SMPs, evaluate how previously proposed mechanisms may or may not be appropriate for SMT, and propose new mechanisms specifically designed to take advantage of unique SMT architectural features. To highlight the potential impact of each of these factors, we present a case study based on the most likely near-term application target for SMT: server software. Because the software architecture of server programs has a strong interaction with each of the design issues we examine, our case study evaluates three different thread models: a multi-process architecture, a multi-threaded architecture, and an event-based architecture.

Overall, our goal is to identify the minimal set of changes that enables applications written for a uniprocessor or a shared memory multiprocessor to achieve good performance on an SMT. Our results demonstrate that SMT-aware memory allocation is essential for good SMT performance and that tuned static memory allocation and synchronization policies raise that level even higher. We find, however, that a few simple changes to the run-time system can provide performance very close to this.

The next section reviews the SMT architecture and examines its effect on memory allocation and synchronization. Section 3 explains the three types of thread models we use for our case study, and Section 4 discusses our methodology. Section 5 presents our case study results, while Section 6 reflects on these results and makes recommendations for SMT runtime support. Section 7 discusses related work and Section 8 concludes.

2 PERFORMANCE IMPLICATIONS OF SMT'S ARCHITECTURE

In this section we first provide a very brief overview of SMT processors. We then examine more closely the relevant differences between SMT, uniprocessors, and SMPs as they affect dynamic memory allocation, stack allocation, and synchronization. To conclude each section we summarize the different policies that we use in our evaluation.

2.1 SMT Overview

SMT is a latency-tolerant processor that fetches, issues and executes instructions from multiple threads each cycle. The processor state for all executing threads, called *hardware contexts*, resides on the CPU, enabling SMT to process instructions from the different threads without resorting to either software or hardware context switching, as is the case for single-threaded uniprocessors or traditional multithreaded processors [1, 2], respectively. This unique feature provides better utilization of execution resources by converting the thread-level parallelism provided by the multiple processor-resident threads into cross-thread instruction-level parallelism. Previous research has established SMT as effective in increasing instruction throughput (i.e., two- to four-fold speedups) on a variety of workloads (including scientific, database, and web servers, in both multiprogrammed and parallel environments), while still providing good performance for single-threaded applications [42, 30, 31, 29].

For our purposes, the most important feature of SMT's architecture is that all contexts dynamically share most processor resources, including the functional units, caches, TLBs, and fetch bandwidth. The sharing of caches, in particular, makes inter-thread data communication and synchronization potentially inexpensive.

2.2 Dynamic Memory Allocation

For a variety of reasons (see Section 3), many server applications make heavy use of dynamic memory allocation, and thus their overall performance can be strongly influenced by allocator design. Important performance factors for an allocator include:

1. Speed – how much time is required to allocate and deallocate a single memory object?

2. Concurrency – to what extent do multiple threads allocate or deallocate memory simultaneously?

3. Data locality – does the allocator induce good or bad data locality for allocated objects within a single thread?

4. False sharing – do allocations by multiple threads result in false sharing (i.e., two threads accessing logically distinct memory that resides in the same cache line)?

5. Fragmentation – how much memory must the allocator request from the operating system in order to satisfy the total memory needs of an application?

6. Thread independence – may objects allocated by one thread be passed to and freed by another thread?

Some of these issues have impact regardless of the underlying hardware architecture. For instance, maximizing data locality and minimizing fragmentation are always beneficial, especially with very large data working sets. Maximizing allocator speed is also important, particularly for servers that allocate and deallocate frequently. Finally, thread independence of memory objects is essential for enabling flexible server design, as argued by Larson and Krishnan [26].

For other memory allocation issues, however, server performance depends heavily upon the underlying hardware. For a uniprocessor, allocator concurrency is of little concern, while on an SMP or SMT, it is essential to performance. Previous research has demonstrated that enabling such concurrency for SMPs depends upon reducing inter-processor data sharing, both true sharing of allocator data structures (e.g., free lists and locks) [26] and false sharing of allocated memory [6]. For an SMT processor, however, the existence of a single, thread-shared cache eliminates the performance penalty of true or false sharing. In fact, prior work has shown that false sharing can improve performance on an SMT [31], because it enables threads to implicitly prefetch data for each other. Consequently, while an SMT processor needs an allocator with good concurrency, using an SMP allocator that reduces inter-processor sharing may be unnecessary or even harmful on an SMT.

To address the various needs discussed above, researchers have designed a wide range of memory allocators; Table 1 shows those that we consider in this work. In the simplest case, *Serial*, only one thread may access the memory allocator at a time. Such allocators may be extremely fast for single-threaded programs [28]. In this work, however, we do not consider it further, since our results show that significant allocator concurrency is essential for good performance on SMT.

The next two allocators, *BaseAlloc* and *FirstFit*, both provide

Table 1: Dynamic memory allocation schemes.

Allocator	Description	In the spirit of...
Single Heap Allocators		
Serial	Only one thread may access the heap at a time.	Solaris, Lea’s Malloc [28]
BaseAlloc	Single heap, but multiple free lists for objects of different sizes.	Compaq Alpha C++
FirstFit	Like BaseAlloc, but will use free list for larger-sized objects if necessary.	Optional with Alpha C++ [12]
Multiple Heap Allocators		
RoundRobin	Each new allocation selects a heap in round-robin order.	MTmalloc [34]
StaticHeap	Each thread is assigned to a unique heap.	Hoard [6], Vee and Hsu [45]
DynamicHeap	Threads initially assigned to a heap, but reassigned after every N allocations.	New to this work

a single heap that contains multiple free lists, where each list is responsible for memory objects of a given size. Because each free list is guarded by its own lock, multiple threads may access the heap simultaneously. BaseAlloc always chooses the free list whose elements are closest to the requested memory size, while FirstFit will consider the free lists of larger-sized objects if the optimal free list is already in use by another thread. BaseAlloc is the standard allocator provided by Compaq's C++ for multiprocessor Alpha systems, while FirstFit is a variant of BaseAlloc specifically intended for “multithreaded applications making heavy use of ... malloc functions.” [12]

The next three allocators use multiple heaps to reduce contention, all modeled after BaseAlloc. Each allocator has a fixed set of heaps, and follows a different policy to determine which heap to choose for an allocation (in our implementation, an object is always freed to the heap from which it originated, so the policy directly affects only allocations). *StaticHeap* assigns each thread to its own heap, while *RoundRobin* always chooses a different heap in round-robin order. We designed a new concurrent allocator, *DynamicHeap,* that combines these features by reassigning a thread to a new heap (chosen in round-robin order) after every N allocations.

To ensure fair comparison, all of our allocators use the same underlying allocation code, but with different heap organizations and options. Our intent is not to select or design the optimal memory allocation algorithm for SMT processors, but rather to identify how important features of any such algorithm interact with the architecture. For instance, both RoundRobin and DynamicHeap are poor choices for SMP allocators, because they actively induce false sharing among threads [6]. Since each heap allocates data to all threads over time, objects within the same cache block may be allocated to threads running on different processors. In addition, these allocators utilize frequently-written global data structures to make heap assignment decisions, a potential coherency bottleneck on SMPs. On an SMT, however, both of these sharing issues may actually improve rather than hinder performance. Section 5 evaluates their actual performance on SMT.

2.3 Static Memory Allocation

The layout of per-thread statically allocated data, such as stacks and global variables, also affects program performance. To some extent, this factor is simpler than dynamically allocated data. Since all allocation occurs just once, at program start-up, allocation speed and concurrency are not a concern. However, because statically allocated data items (particularly stacks) are accessed frequently, their layout can impact overall cache behavior.

On a uniprocessor, compilers and application designers try to maximize intra-thread spatial and temporal locality by placing allocated data items that are used together contiguously in memory. For an SMP, the additional need to avoid false sharing between threads on different processors leads to the use of data restructuring techniques, such as group and transpose, indirection, and padding data structures to fill out cache lines [20, 3].

On an SMT processor, while inter-thread locality is important, false sharing is not. Instead, SMT processors face potential performance problems when the data structures of different threads or processes conflict in the cache. For instance, Lo et al. [29] identified stack conflicts as a significant performance problem in a multi-process database workload. In that case, conflicts arose because the stack of each process resided at the *same* virtual address, and thus conflicted in the virtually-indexed, level-1 cache[1]. In our evaluation we identified a related but new source of stack conflicts for multithreaded applications that arises despite stacks having *different* virtual addresses. Section 5 discusses the causes of these conflicts and evaluates how stack offsetting can reduce them.

2.4 Synchronization

Factors that affect the relative performance of synchronization techniques include:

1. Overhead – what is the cost to acquire and release an uncontended lock?

2. Contention – how does synchronization performance vary as multiple threads contend for the same lock?

3. Performance interaction – how does the behavior of one thread processing synchronization operations affect the execution of other threads?

4. Scheduling – how do the synchronization mechanisms communicate with the operating system and/or run-time thread schedulers?

The impact of all these factors depends upon the underlying hardware architecture. On an SMP, for instance, in-cache spinning on a lock variable may offer low-overhead synchronization when contention is low, but generate performance-degrading coherence traffic when contention is high [16]. Consequently, researchers have proposed techniques, such as queue-based locks [22] to reduce bus traffic, and data co-location [22] to reduce overall critical section time. On a single SMT processor, techniques to

[1] Such conflicts can also occur on an SMP, but have little performance effect, because only one process utilizes the cache at any instant in time.

Table 2: Synchronization primitives. For this work we configure SW-spin-then-block and HW-lock-then-block to wait for approximately 1500 cycles, about the amount of time needed to perform a context switch, before blocking.

Synchronization Primitive	**What to do if lock is unavailable**	Spin?	Block to OS?
SW-block	Block immediately to the OS.	no	immediately
SW-spin	Repeatedly spin in software until lock is available.	yes	no
SW-spin-then-block	Spin up to K times, then block to OS if necessary.	yes	after K spins
HW-lock	Block in hardware until lock is available.	no	no
HW-lock-then-block	Block in hardware for up to T cycles, then block to OS if necessary.	no	after time-out of T cycles

decrease inter-processor bus traffic are unnecessary. However, synchronization primitives that rely on spinning may still degrade performance, since a context that passes time by spinning consumes execution resources that may be better used by other contexts [38]. Consequently, SMT researchers have proposed a hardware *lockbox* that blocks a waiting thread, preventing it from utilizing any hardware resources until it acquires the needed lock [44, 11]. Although synchronization is a more important issue on these processors, it is necessary even on a uniprocessor to guarantee mutual exclusion between threads in the face of context switches and interrupts.

Furthermore, when the number of executing threads is greater than the number of processors or hardware contexts, lock mechanisms must also consider how to interact with various thread schedulers. On a uniprocessor, the decision is trivial; if a lock is not available, the thread should immediately block, since the thread holding the lock cannot release it unless it is given the chance to execute. For an SMP, a better strategy is to spin for a short period of time before blocking. Deciding how long to spin involves a trade-off between minimizing context switch overhead and maximizing useful work.

For an SMT processor, these decisions are complicated by the lockbox, which affects thread scheduling in two ways. First, blocking a thread in hardware prevents it from degrading the performance of other executing threads, but excludes another thread from that context. Thus, an important extension may be to augment hardware blocking with a time-out. This enables a new type of synchronization mechanism, *HW-lock-then-block*, which is analogous to the software-only *Spin-then-block*. Second, the lockbox affects thread scheduling by arbitrating among different threads competing for a lock. For instance, the lockbox may grant lock requests in FIFO order, to preserve fairness. Alternatively, favoring waiting threads that are still context-resident (instead of swapped out by the operating system) may improve performance by reducing context switch overhead.

In this work, we evaluate the five synchronization mechanisms shown in Table 2. *SW-block* is the default mechanism provided on Tru64 UNIX (implemented as `pthread_mutex_lock()`), which serves as our baseline. *SW-spin* and *SW-spin-then-block* represent reasonable synchronization mechanisms for an SMP processor, while *HW-lock* and *HW-lock-then-block* attempt to improve upon these mechanisms by eliminating spinning.

3 SERVER SOFTWARE ARCHITECTURE

Server applications, including web servers, search engines, and databases, represent an increasingly important class of commercial applications. Previous work has shown that SMT is particularly effective on these workloads, because a server's multiple outstanding requests provide a natural source of thread-level parallelism for hiding cache latencies [39, 29]. Furthermore, modern server applications are likely to have very high rates of memory allocation (and thus of synchronization within the allocator as well), both because they are often written in an object-oriented style and because the unpredictable nature of requests and responses require flexible allocation of memory on demand [26].

To achieve high performance, a server application must be written in a way that exposes the workload's natural parallelism. Traditionally, servers use either a *multi-process* (MP) or a *multi-threaded* (MT) thread model (see Pai et al. [35] for an overview). In the multi-process approach, each incoming request is assigned to a process with its own distinct address space, resulting in little if any direct interaction between separate processes. In the multi-threaded approach, each incoming request is assigned to a unique thread, but all threads share a common address space, which causes some degree of interaction between the threads (because they share memory) and enables software optimizations, such as a thread-shared software result cache. More recently, researchers have proposed using an *event-based* software architecture for server systems, both for improved robustness and fairness [46] and for improved performance [27]. Like a multithreaded software architecture, these event-based approaches use a shared address space, but decompose each request into a number of events that are multiplexed among a small number of threads to minimize context-switching costs. This decomposition exposes similar computations across multiple client requests (threads), allowing the construction of intelligent policies that improve locality and performance by scheduling related events both consecutively and, on an SMT, simultaneously [27, 32]. Consequently, unlike the previous two approaches, the threads in an event-based server have a high degree of interaction.

Choosing the appropriate thread model for a server is a complex decision influenced by many factors including performance constraints, software maintainability, and the presence of existing code. In this work we evaluate all three thread models. Our intent is not promote any model over the other, but to use the different characteristics of each model to examine the extent to which the interaction between different threads or processes impacts memory allocation and synchronization on SMT.

4 METHODOLOGY

4.1 Simulator

Our SMT simulator is based on the SMTSIM simulator [43] and has been ported to the SimOS framework [39]. It simulates the full pipeline and memory hierarchy, including bank conflicts and bus contention, for both the applications and the operating system. The simulation of the operating system enables us to see OS effects on synchronization overhead and memory allocation.

Table 3: SMT configuration parameters.

CPU	
Thread Contexts	4
Pipeline	9 stages
Fetch Policy	8 instructions per cycle from up to 2 contexts [42]
Functional Units	8 integer (including 4 Load/Store and 1 Synch. 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 [33]
Local Predictor	4K-entry prediction table indexed by 2K-entry history table
Global Predictor	8K entries, 8K-entry selection table
Branch Target Buffer	256 entries, 4-way set associative
Cache Hierarchy	
Cache Line Size	64 bytes
Icache	64KB, 2-way set associative, single port
	2 cycle fill penalty
Dcache	64KB, 2-way set associative, dual ported. Only 1 request at a time supported from the L2
	2 cycle fill penalty
L2 cache	16MB, direct mapped, 20 cycle latency, fully pipelined (1 access per cycle)
MSHR	32 entries for the L1 caches, 32 entries for L2
Store Buffer	32 entries
L1-L2 bus	256 bits wide, 2 cycle latency
Memory bus	128 bits wide, 4 cycle latency
Physical Memory	512MB, 90 cycle latency, fully pipelined

The configuration for our experiments is shown in Table 3. The most important parameters for this study are the number of contexts and the size of the L1 caches. Most machine parameters are similar to the previously planned Alpha 21464 [36].

Our studies focus on CPU and memory performance bottlenecks. In the interest of simulation time, we simulate a zero-latency disk, modeling an in-core database. This choice does not unduly influence our results, because the total working set of our application (around 150 MB) easily fits within main memory.

4.2 Workload

For our case-study, we constructed three different versions of *htdig* [18], a popular search engine typically used to index and search a single site. This choice is appropriate for studying servers, because it has significant amounts of concurrency, is based on an object-oriented design, and exercises the operating system for both disk and network I/O. To create the multithreaded and multi-process versions, we made minor changes to the original CGI program to enable it to handle multiple requests simultaneously, using either threads or processes. For the event-based version, we manually decomposed the search process into 11 "stages" and constructed the application using an event-based application toolkit [32]; the thread scheduling policy was designed to encourage the simultaneous execution of related events to improve cache locality.

Our case-study server stresses both memory allocation and synchronization. The server requires an average of 15,000 memory allocations per request, causing it to spend about 36% of its time in the allocator. This represents a very allocation-intensive workload, but not unreasonably so[1]. In addition, synchronization in our application is dominated by locks in the memory allocator. Each request requires approximately 27,000 lock operations, 88% of which are due to memory allocations. Locks are held for an average of 125 instructions.

We configured the MT and MP servers to use eight threads or processes, respectively, because we found that this was the minimum necessary to ensure that the processor was fully utilized. In order to minimize context switching overhead and provide maximum flexibility for the event scheduler, we configured the event-based server with just four primary threads (the same as the number of SMT contexts). For the multiple-heap allocators, we used twice as many heaps as threads (as is done in Berger et al. [6]).

Due to the lack of standard benchmarks for search engine performance, we constructed our own workload based on actual searches related to web pages in our department. We first used *htdig* to construct a searchable database of all the HTML content in our department. Next, we examined a daily departmental web log to find all Google search queries that resulted in a page view at our site, yielding a set of 1711 queries. To enable tractable simulation time, we eliminated all queries that required more than three times the average number of instructions to execute. This eliminated 79 (4.6%) of the queries. Simulations run with the full query set vs. the reduced query set showed that the overall performance impact of this change was only 2%.

4.3 Metrics

Server performance is often reported in terms of raw throughput (e.g., the number of requests processed over some time period). For two reasons, however, this metric does not accurately reflect performance for our experiments. First, not all requests represent an equal amount of work. For instance, in our workload, the most expensive requests require more than 30 times as many instructions as the least expensive. Second, not all request processing completes during the simulation. For instance, a request may be largely processed during the simulation, but actually complete just after the experiment ends (and thus not be counted by a raw throughput measure). Despite our comparatively long simulations (in the billions of cycles), we execute at most hundreds of requests and thus both of these effects can be significant.

We instead report results in terms of *normalized request throughput*, which is calculated as follows. We first warm-up the server by executing until the server has completed 200 requests, and then simulate in detail for 2 billion cycles. Within this time, we define a 1-billion cycle experimental window starting at 500 million cycles. To limit simulation time, results for the sensitivity analyses in Section 5.5 were based on a window size of 500 million cycles.

Normalized throughput is then calculated as the sum, for all requests, of the fraction of each request that was processed during the window, multiplied by the weight of the request[2]. Weights are

[1] For instance, Berger et al. [8] found that a range of applications spent 0% to over 40% of their execution time performing allocator operations (with an average of 16%), even though they used optimized custom allocators.

based upon the number of instructions required to execute each request on a single-threaded superscalar. We found that this metric produced consistent results (within 2%) for repeated simulations on different portions of the query input set.

5 CASE-STUDY RESULTS

We first present data for the multithreaded server model, comparing it to the multi-process server. After drawing conclusions from these results, we then present data for the event-based server that both confirm the conclusions and highlight how different thread models can stretch an SMT processor in other ways. Finally, we provide sensitivity analyses that examine the extent to which our results depend on particular memory allocation parameters.

5.1 Effects of Dynamic Memory Allocation

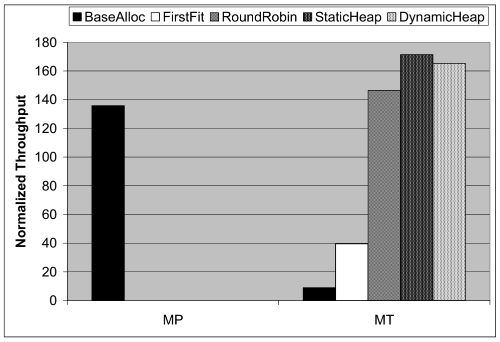

Figure 1. Server request throughput when varying dynamic memory allocation strategies. The MP server uses only BaseAlloc, since processes do not share dynamically allocated memory.

We first examine dynamic memory allocation, since our results indicate that it has the most significant effect on performance. Figure 1 shows request throughput for the multithreaded (MT) server for each of the five dynamic memory allocators discussed in Section 2.2. Since each process in the MP server has its own address space (and thus its own private allocator), there is no need to change its allocation strategy; instead, we show MP results using just the baseline allocator, BaseAlloc, a common choice for SMPs. All experiments use the default synchronization and static memory allocation schemes, SW-block and no offsetting, and are executed on the SMT simulator.

The results demonstrate that on SMT the MT server has very poor performance with single-heap allocation (BaseAlloc and FirstFit), compared to MP, achieving only 7% to 29% of MP's request throughput. However, using any of the multiple-heap allocators improves MT's performance by more than an order of magnitude, enabling MT to outperform MP by 8% to 26% for our benchmark.

The single-heap allocators perform poorly on the MT server because a single heap does not provide enough allocator concurrency to fully utilize a processor such as SMT. Instead, threads often wait to access the allocator, leading to frequent context switches and idle time. For BaseAlloc, this problem is so severe that the server spends only 17% of its time in user-level code (where almost all search engine activities take place), compared to a minimum of 86% for the multiple-heap allocators. At first glance, this result may seem surprising, given BaseAlloc's use of multiple, independent free-lists. However, previous research has found that programs often allocate the vast majority of all objects from just a few object sizes [21], which greatly decreases the utility of multiple free-lists. Our own experiments confirmed this, showing that about 50% of all allocations in BaseAlloc involved the same free-list, and 75% involved just two free-lists. FirstFit's more flexible choice of a free list reduces allocation contention, increasing performance more than four-fold, compared to BaseAlloc. Contention still remains significant, however, because, even though FirstFit always has a choice of free lists when allocating a new object, deallocations must return objects to their originating list. If the original list is already in use, contention occurs.

In contrast, the multiple-heap approaches greatly increase allocator concurrency by eliminating the single-heap bottleneck. Consequently, all multiple-heap allocators see vastly improved performance over BaseAlloc and FirstFit. Within these allocators, however, secondary performance effects remain. Most significantly, RoundRobin suffers from poor data locality, because consecutive allocations from the same thread are likely to come from different heaps. This effect causes the miss rate in SMT's thread-shared data TLB to increase from 0.2% for StaticHeap to 1.5% for RoundRobin; likewise, in the thread-shared data cache the miss rate increases from 8.4% to 11.9%. Consequently, StaticHeap outperforms RoundRobin by 17%.

Despite also allowing threads to allocate memory from multiple heaps over time, DynamicHeap preserves data locality by making multiple allocations before switching to another heap. As a result, it has data TLB and cache miss rates of just 0.6% and 9.3%, respectively, and outperforms RoundRobin by 14%. DynamicHeap's heap assignment policy does, however, slightly increase contention for new allocations compared to StaticHeap, leading to a small 4% performance difference. The next section and Section 5.4 revisit this issue, exploring ways of mitigating this contention and pinpointing situations in which DynamicHeap's different allocation style actually leads to a performance gain.

Finally, the multi-process server's performance is lower than that of the multiple-heap allocators with MT, because its separate address spaces eliminate much of the potential for sharing in SMT's L1 data cache. This produces an average memory access delay that is 31% to 79% higher than the MT multiple-heap allocators, and thus an overall IPC that is 8% to 19% lower.

In summary, even on an SMT processor where sharing is beneficial and synchronization requires no inter-processor communication, the most aggressive single-heap allocators still provide too little concurrency for a multithreaded application with demanding amounts of memory allocation. On the other hand, multiple-heap allocators with large amounts of concurrency produce good performance. This performance can be marred, however, by a loss in data locality induced by an allocator with no affinity between threads and heaps. (Section 5.5 examines both of these issues in more detail to evaluate how much thread/heap affinity and what level of allocator concurrency are necessary to ensure good performance.) Finally, a multi-process thread model can effectively use the sim-

[2] We utilize request completions (i.e., a client request was processed) rather than an IPC-based metric (as in Tullsen and Snavely [40]), because, for our workload, lock spinning or operating system activity may artificially inflate thread IPC.

pler single-heap memory allocator, but it performs less well than a well-configured multithreaded application.

5.2 Effects of Synchronization Strategies

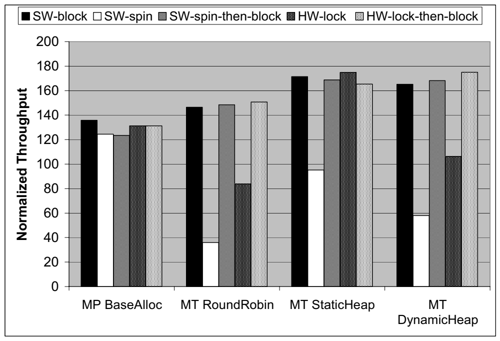

Figure 2. Server request throughput from varying the synchronization policy for the MP and MT servers.

The previous section demonstrated that the choice of memory allocator is a crucial factor affecting the performance of a multithreaded server executing on an SMT processor. Now that we have identified several memory allocators that alleviate this potential bottleneck, we investigate the extent to which synchronization affects the performance of the improved system.

Figure 2 shows the results from evaluating synchronization policy, using the mechanisms in Table 2 and the best MT and MP memory allocators. Within in each group, the first bar is the performance for the baseline synchronization mechanism, SW-block, and is thus equivalent to the results from Figure 1. The next two bars are strategies that involve spinning, while the final two use SMT hardware locks.

Lock contention in our server is primarily due to memory allocation. Since the MP server has no allocator contention, its results reflect the direct, uncontended overhead of the synchronization policies, while the MT results show performance under various degrees of thread contention.

The MT results highlight (1) the downside of spinning and (2) the importance of blocking to the operating system. As expected, Spin-only has very low performance, because it steals hardware resources from threads in other hardware contexts and prevents other runnable threads from context switching into its own. Spin-then-block, however, has surprisingly good performance, in fact, comparable to the best strategies that use hardware locks, despite its initial spinning. This is because the extra overhead of spinning is small compared to the benefits of relinquishing the SMT context to a thread that can do useful work.[1]

HW-lock, on the other hand, has very unpredictable performance, sometimes outperforming all other synchronization schemes (with StaticHeap), sometimes doing considerably worse than the best strategy (39% worse with RoundRobin and 52% with DynamicHeap). Because HW-lock never communicates with the operating system, it can potentially be blocked for a very long time if waiting for a lock that is held by a thread not currently scheduled. We have observed pathological cases in which a hardware-locked thread prevents the thread holding the lock from executing for hundreds of millions of cycles. While the astute programmer may be able to avoid these situations with careful thread management, a much better choice is to use HW-lock-then-block. This mechanism both eliminates spinning and yields to the operating system scheduler when necessary, thus producing good results for StaticHeap and the best results for RoundRobin and DynamicHeap (where contention is more of an issue than with StaticHeap).

To some extent, our results may underestimate the performance benefits of hardware locks without blocking. In our application, the distribution of allocation requests across multiple heaps significantly reduces lock contention. With multiple-heap allocators, virtually all locks are already available when requested, and are held for a significant period of time (relative to lock acquisition) before they are released. For non-server applications, such as loop-based scientific programs, that have more lock contention and shorter critical sections, lock acquisition overhead is more significant and hence the impact of using spinning vs. efficient hardware locks is much larger [44].

Finally, note that optimizing the synchronization strategy had a small but noticeable effect on the relative performance of the different dynamic memory allocation schemes. With the baseline synchronization strategy used in Figure 1, we found that DynamicHeap lagged behind StaticHeap, because it slightly increased contention. Figure 2, however, demonstrates that improving the synchronization strategy to HW-lock-then-block enables DynamicHeap to match the performance of the best-case for StaticHeap. Section 5.4 considers the relative merits of StaticHeap vs. DynamicHeap in more detail.

Overall, our results demonstrate that, for a multithreaded server application where dynamic memory allocation is the dominant performance factor, proper allocator design may reduce lock contention enough that the *efficiency* of the synchronization mechanism is a second-order effect, even on an SMT, where primitives based on spinning can delay non-spinning threads. Instead, to obtain the best performance, the key synchronization requirement is the *ability to yield to the OS thread scheduler* for scheduling decisions, whether spinning or hardware-blocking. Consequently, if SMT hardware locks are used, they must provide a time-out feature to enable this interaction. However, that being said, for both multi-process and multithreaded servers, many of the better synchronization alternatives have roughly comparable performance, thus affording SMT programmers and hardware designers considerable leeway in choosing a synchronization mechanism.

5.3 Improving Static Memory Allocation

As discussed in Section 2.3, the default stack layout produces cache conflicts for the multi-process thread model, because each process stack uses the same starting virtual address. In addition, our experiments identified a (related) source of stack conflicts for multithreaded applications on SMT. We found that for all MT server alternatives, the threads' stacks completely overlapped in the L1 cache, despite having different virtual addresses. The conflicts arose because the operating system created each new stack precisely 64 KB apart from the previous stack. Consequently, our virtually indexed, 64 KB L1 data cache experienced very frequent conflict misses.

[1] We utilized simple hand-coded spin locks. Careful tuning might achieve some performance gains, but would not affect the relative importance of blocking to the operating system.

Table 4: Using application-level offsetting to reduce data cache miss rates and improve overall performance.

Server Configuration	Original Behavior		With Offsetting		Throughput Improvement
	L1 Data Miss Rate	Throughput	L1 Data Miss Rate	Throughput	
MT StaticHeap, SW-spin-then-block	8.4%	167.6	4.4%	178.8	7%
MT StaticHeap, HW-lock-then-block	8.7%	166.9	4.7%	182.9	10%
MT Dynamic Heap, SW-spin-then-block	9.5%	170.7	5.0%	183.1	7%
MT DynamicHeap, HW-lock-then-block	8.8%	174.1	4.8%	184.3	6%
MP BaseAlloc, SW-block	14.3%	130.9	10.9%	135.4	3%

The conflict misses can be avoided using *application-level offsetting,* similar to that used by Lo et al. [29], which shifts the stack of each thread by the page size times the thread creation order. In our case, we shifted each of the eight threads' stacks, so that each stack mapped to a different eighth of the L1 data cache. Table 4 shows the performance impact of using this offsetting to reduce cache conflicts on SMT. We show results for the MP server with the baseline allocation and synchronization schemes, and for the MT server with the two best allocation and synchronization schemes from the previous sections. The last column represents the total performance gain from using offsetting compared to the baseline behavior.

The data shows that the improved stack layout reduces cache misses; for instance, the L1 data miss rate decreased from an average of 8.9% to an average of 4.7% for the multithreaded servers, and from 14.3% to 10.9% for the multi-process server. Consequently, overall performance improved by between 6% and 10% for MT and 3% for MP. Note that the performance improvements would have been greater had our caches been direct-mapped; instead, the 2-way set associativity permitted related cache lines from up to 2 threads to reside in the cache simultaneously.

Thus, while the default stack layout policies for a multiprocessor operating system may easily induce performance problems for either multi-process or multi-threaded servers on SMT, simple, cache-conscious techniques to reduce stack conflicts can improve the situation for both thread models.

5.4 Results for the Event-Based Server

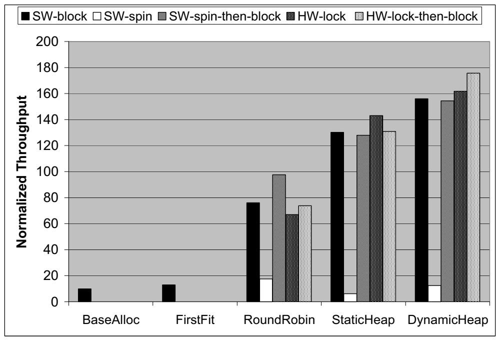

Figure 3. Server request throughput for the event-based server, varying the dynamic memory allocation and synchronization schemes, and without application offsetting.

The previous sections identified a number of important guidelines for dynamic and static memory allocation and synchronization for multithreaded and multi-process SMT servers. We now validate and explore these findings with an event-based version of the same server. Because the event-based server is also based on threads, we expect to find similar trends. Nonetheless, threads in an event-based server interact more frequently than those in the other thread models, and thus may stress the machine in different ways.

Figure 3 contains results for the event-based server for each of the five dynamic memory allocations strategies, without application offsetting. For the three multiple-heap allocators, we also show results from varying the synchronization strategy. (All synchronization schemes performed equivalently with single-heap allocation.). The results confirm that the single-heap allocators (BaseAlloc and FirstFit) are insufficient for MT or event-based servers, and that the performance of RoundRobin lags behind the other multiple-heap allocators, because it suffers from poor data locality. In addition, we see some of the same general trends for synchronization: spinning has very poor performance, and limited spinning (Spin-then-block) performs quite well. However, unlike the MT experiments, HW-lock always has reasonable performance; because the event-based experiments execute with only four major threads, there is no need to block to the operating system. Other experiments with eight threads (data not shown) more closely match the MT results.

Unlike with MT, the DynamicHeap allocator in the event-based thread model obtained significantly higher request throughput than StaticHeap. StaticHeap's performance suffers from contention when one thread allocates a number of objects which are then passed to and freed by other threads. This type of producer-consumer behavior was identified by the StaticHeap-like Hoard [6] as producing their worst-case behavior. In essence, allocators like StaticHeap guarantee that allocations, but not necessarily deallocations, by different threads do not conflict. In contrast, DynamicHeap permits some conflicts for allocations in order to reduce deallocation conflicts. This effect is particularly important for event-based servers, which execute identical or related pieces of code (such as resource deallocators) simultaneously. In particular, with the HW-lock-then-block mechanism, the StaticHeap allocator generated about 64,000 heap conflicts per 100 million cycles, of which 72% were caused by simultaneous deallocations by different threads. Using DynamicHeap distributed the deallocations more evenly across heaps, reducing the total number of heap conflicts to 34,000, a reduction of 47%. Consequently, DynamicHeap outperformed StaticHeap by 13% to 34% (excluding the outlier SW-spin).

Finally, we also evaluated the extent to which static data layout was a performance issue for the event-based server. We found that although applying application-level offsetting always helped data cache miss rates (on average reducing them from 8.0% to 6.2%), overall performance varied from +7% to -9%. Offsetting may affect factors other than cache performance; for instance, it may increase data TLB misses if the new stacks are not aligned on page boundaries. Normally these other effects are tempered by the decrease in cache misses. However, in our implementation of the event-driven server the threads' stacks were not allocated in a deterministic order relative to other memory objects. Accordingly, the precise layout of the threads' stacks was non-deterministic, as it was affected by the precise timing of allocations and the order in which threads executed during initialization. Consequently, the original program's thread stacks did not completely conflict in the stack. As a result, there was reduced benefit to the offsetting, and other factors sometimes dominated.

In summary, our event-based server results confirm the general conclusions we drew from the MT and MP servers: single-heap allocators introduce too much contention, and synchronization primitives must eventually yield to the operating system to allow other threads that have useful work to be done to be scheduled. While allocation conflicts are an important effect to minimize, deallocation conflicts can also be a significant performance issue for applications (such as our event-based server) that pass objects between threads.

5.5 Sensitivity Analyses

In this section we vary some of the parameters used to configure our dynamic memory allocation mechanisms in order to investigate the sensitivity of our results to these features. These results all use the most robust allocator we found in the previous sections, DynamicHeap, along with the best synchronization strategy, HW-lock-then-block.

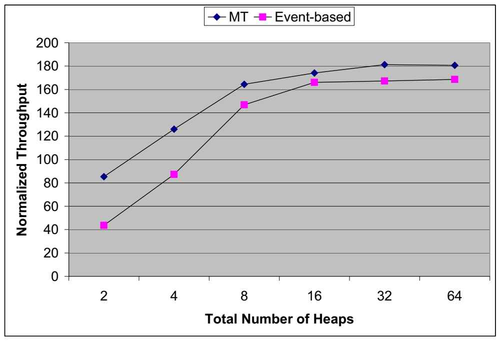

Figure 4. Results using the DynamicHeap allocator while varying the total number of heaps. The previous results assumed that the number of heaps was twice the number of threads, i.e. 16 heaps for MT, 8 for Event-based.

Section 5.1 demonstrated that multiple heap allocators were necessary to provide enough concurrency for our application. In the original experiments, the allocators used twice as many heaps as threads. Figure 4 explores the sensitivity of request throughput to the heap/thread ratio by varying the total number of heaps used by the DynamicHeap allocator. We find that throughput steadily improves as the number of heaps increases, though most of the performance benefit is achieved when using only eight heaps. Recall that DynamicHeap's periodic reassignment of heaps to threads results in allocations and deallocations being distributed across all heaps. Since we simulate a processor with four contexts, using four heaps will most likely give rise to inter-thread conflicts with most allocations and deallocations. Doubling the number of heaps (to eight) decreases these conflicts and doubles performance, but the returns diminish with larger numbers of heaps.

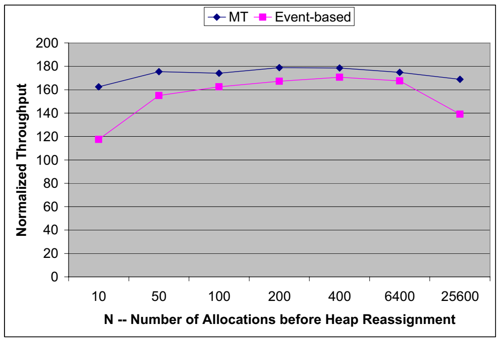

Figure 5. Results using the DynamicHeap allocator while varying the number of allocations before a thread is assigned to a new heap.

Figure 5 shows the performance of the DynamicHeap allocator as a function of N, the number of allocations before a thread is assigned to a new heap. For both MT and Event-based, performance is fairly constant for a wide range of N from 50 to 6400. For very small N, it drops for the same reason that the performance of RoundRobin suffered: each thread keeps the same heap for only a very short period of time, decreasing data locality. Likewise, as N grows very large, performance starts to approximate the behavior of the StaticHeap allocator (the two are equivalent for N = infinity). Request throughput begins to fall, more significantly for the event-based server. (Results in Figure 3 showed that the fixed heap assignments of StaticHeap were more of a problem for this type of server.)

Overall, we find that the performance of the multithreaded and event-based thread models with DynamicHeap depends upon the appropriate choice of both the number of heaps and the number of allocations to delay before changing heaps, but that a very wide range of choices produces near-ideal performance.

6 PUTTING IT ALL TOGETHER

6.1 Designing SMT-friendly Server Support

Our results have shown that support for threaded server software for SMT borrows mechanisms from both uniprocessors and shared memory multiprocessors, and occasionally requires techniques tailored specifically to its unique hardware capabilities.

Allocation concurrency is the dominant performance factor for our workload when executing on an SMT, just as it would be on an SMP. Improving synchronization is fruitless as long as contention within memory allocation serializes most execution time. A multi-heap allocator solves this, but must be careful not to destroy the locality of allocated data. Although SMT is a tremendous latency-hiding vehicle, our data showed that the large number of intra-thread misses seen in the RoundRobin allocator could not be easily absorbed by the thread-shared data TLB and data cache. Instead, the heap assignment strategy of DynamicHeap is better for SMT, because it distributes both allocations and deallocations across multiple heaps while still preserving data locality. We found that such a strategy performed comparably or better than StaticHeap or RoundRobin for all of our experiments.

Once the allocator-induced contention has been eliminated, the most relevant factor for SMT server performance shifts to the design of the synchronization primitive, and specifically, to its interaction with the operating system's scheduler. OS intervention is crucial for both software- and hardware-based locking. On the software side, blocking to the operating system prevents spinning threads from stealing SMT hardware resources from those that are executing useful instructions. On the hardware side, it insures that hardware-blocked threads do not impede unblocked but descheduled threads from running; the operating system can also make better thread scheduling decisions than SMT hardware, because it has global knowledge of which threads can do useful work. In addition, our HW-lock-then-block synchronization strategy can be thought of as an application of scheduler activations [4]. HW-lock-then-block adds an interface between the scheduler that is implicit in the SMT lockbox and the scheduler in the operating system, whereas scheduler activations add an interface between the OS and user-level thread schedulers.

Our work extends earlier results that inter-process stack interference can be a problem for multi-process workloads to multithreaded servers. Stack offsetting is an important optimization for servers executing on SMT, since the stacks of threads or processes that are executing concurrently may conflict in the cache, reducing overall cache performance. The technique can also be useful for event-based servers, but must be applied more carefully.

Specific memory allocation and synchronization alternatives, DynamicHeap and HW-lock-then-block, are necessary for the very best performance on SMT, for both the multithreaded and event-based thread models. However, there is a wide range of strategies that will achieve very good performance. SMT hardware designers and threaded application developers therefore can choose a strategy based on criteria other than high-performing server code (e.g., good performance for fine-grain synchronization found in non-server workloads).

Our results also have interesting implications for a server constructed of multiple SMT processors within a single system. This "SMP of SMTs" presents a new set of challenges that requires further research. We speculate, however, that such a system should use synchronization and static memory allocation strategies that work well for a SMT processor (e.g. HW-lock-then-block and stack offsetting), since these do no harm on a multiprocessor system. As noted previously, however, the DynamicHeap allocator is poorly suited for a SMP. In this case the choice of the ideal allocator is more complex, favoring SMP-centric allocators where the number of processors is large, a SMT-centric allocator where producer-consumer behavior is significant, or perhaps a hybrid combination of the two approaches.

6.2 Run-time Support

Although most of our techniques could be implemented within applications, we believe that a better place to implement these improved mechanisms is in application-level run-time support software (such as libC, or the pthreads library). This strategy encourages portability across architectures and operating systems, prevents authors from having to tune their software to specific architectural details, and encapsulates these mechanisms in one software layer, making it possible to tune the mechanisms to affect all applications that use them.

It is common for the same server application code to run on both uniprocessors and SMPs; this code could also be used on SMTs. Providing SMT-appropriate run-time support ensures good performance even if an application was not specifically tuned for SMT. But even for SMT-targeted applications, the application writer may not know important run-time details, such as the number of contexts that will be allocated to an application. SMT-aware run-time support can learn this information from the operating system and adjust accordingly, for example, alter the number of heaps used for memory allocation, offset thread stacks to avoid cache conflicts, or change the synchronization scheme (e.g., to never block to the operating system if it detects that the total number of threads is less than or equal to the total number of contexts). Finally, including appropriate memory allocation and synchronization strategies in the run-time system saves programmers from having to reinvent the wheel for each application. Programmers may still provide their own mechanisms where appropriate, but providing reliable primitives for the common case makes good design sense.

7 RELATED WORK

Prior research on multithreaded processors has considered mainly multiprogrammed [43, 15, 40] or scientific workloads [15, 30, 25] such as the SPLASH-2 benchmarks [47], rather than server applications. Evaluation of server applications, however, is particularly important, because several studies [5, 13, 39] have found that the performance of servers is often far worse than that of typical SPEC integer and floating point applications on both uniprocessors and SMT processors. A few studies [29, 39] have considered database and web server applications and found that SMT can provide significant speedups compared to a uniprocessor. These studies, however, only considered multi-process workloads, where the memory allocation and synchronization issues that we consider are much more straight-forward than for multithreaded applications.

Lo et al. [31] reconsiders several compiler optimizations designed for shared memory multiprocessors and suggests ways that they should be applied differently on an SMT processor. Our work on SMT also reevaluates some techniques drawn from SMPs, but focuses on techniques that can be implemented in the runtime system with no changes to the application or compiler. In addition, a small body of research has considered improvements to the operating system for SMT processors. For instance, Snavely and Tullsen [40] propose using sampling to determine mixes of threads that execute well together, while Redstone [38] investigates the performance impact of eliminating spinning synchronization from *within* the operating system. These techniques would complement our work on improving runtime support for multithreaded and multi-process applications.

For dynamic memory allocation on multithreaded processors, we are aware only of the superthreaded work of Tsai et al. [41]; they use (but do not evaluate) multiple heaps to reduce allocator contention on a SMT-like machine. A large body of previous work, however, has considered this problem for shared memory multiprocessors. Most work has focused on reducing allocator contention by parallelizing a single heap [9, 19], generally by using multiple free lists (similar to our baseline allocator provided by Compaq). Our results confirmed that these approaches often fail to scale well, because many programs allocate objects from only a few different object sizes [21]. Alternatively, researchers have considered reducing contention with multiple heaps [26, 6, 45]. While using multiple heaps reduces lock contention, sharing and locality effects among these heaps can have a serious effect on performance, issues addressed by LKmalloc [26] and Hoard [6]. The StaticHeap that we considered in this work has very similar behavior to the best SMP allocator we found, Hoard. However, Hoard also uses a global heap to redistribute memory among per-thread heaps when necessary, an important feature for a complete implementation of any of the multiple-heap allocators that we consider.

Haggander and Lundberg [17] recommend using a multi-process architecture where possible to avoid the possible pitfalls of multithreaded memory allocation. Our work demonstrates that this option is indeed simpler for workloads where little interaction is needed between different requests, but suffers a performance penalty. Alternatively, many applications (e.g., Apache and several SPEC benchmarks, see [8]) attempt to improve multithreaded memory allocation by the use of application-specific memory allocators. Recent research proposes the use of *heap layers* to make such allocators more reusable and maintainable [7].

Our evaluation of dynamic memory allocation differs from previous work in several ways. First, we present the first experimental results of which we are aware that demonstrate the performance impact of memory allocation for a multithreaded processor. Second, we examined how the memory allocator interacts with the unique features of an SMT processor, particularly the shared TLB and data caches. Finally, we demonstrated how a few simple changes to existing allocators based on these features could produce significant performance improvements, eliminating the worst case behavior of the best known SMP allocator (Hoard) for the common producer-consumer software pattern.

Our evaluation of the layout of statically allocated data extends the earlier work by Lo et al. [29] to multithreaded applications. Kerly [24] also notes the possibility of multithreaded stack interference on an Intel hyperthreaded processor and suggests similar techniques for reducing the conflicts. Kerly, however, considers this problem to be temporary and expects hardware features to address it in the future; we argued instead that the runtime system is well-suited for tackling the problem, and measured the potential impact of this interference for three different types of software architectures.

Section 2.4 discussed the previous research that exists for SMPs and, to some extent, SMT processors, regarding synchronization. Because earlier work either did not fully simulate the operating system [29, 11] or used only as many threads as contexts [44], previous SMT synchronization research focused upon eliminating spinning rather than communicating with the operating system. We demonstrate that operating system interaction is likely to be paramount, particularly for server applications, which typically have an abundance of independent threads. Our work extends the classical spin-then-block software algorithms investigated by Boguslavsky et al. [10] and Karlin et al. [23] to a hardware wait-then-block technique best suited for SMT.

Intel [14] also discusses the need for threads on a multithreaded processor to block to the operating system, but they provide no way to combine this with efficient hardware locks. Other relevant work includes *speculative lock elision* [37], which improves performance by dynamically eliminating the need for many synchronization operations rather than by improving synchronization.

8 CONCLUSION

In this paper, we examined how dynamic memory allocation, stack allocation and synchronization techniques affect the performance of multithreaded server software on SMT processors. Although significant previous work exists in these areas, we found that a re-examination was necessary, because SMT changes key hardware features found on both uniprocessors and multiprocessors. Consequently, optimizing multithreaded software performance on SMTs required borrowing some techniques from uniprocessors, borrowing others from multiprocessors, and inventing new SMT-specific strategies.

For our workload, we found that eliminating contention in the memory allocator was the most important optimization. With this in place, several second-order optimizations were possible, including: picking a synchronization strategy that could communicate with the OS scheduler, optimizing the layout of allocated objects to be locality aware, and eliminating allocator contention during deallocations. Overall, our techniques were able to increase the performance of a naive multithreaded implementation dramatically. We argued that most of these benefits could be realized by implementing the optimizations within the run-time support libraries. Doing so would increase the portability of applications, as programmers would not need to embed architecture-specific optimizations in their applications.

9 REFERENCES

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *17th Annual International Symposium on Computer Architecture*, June 1990.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, June 1990.

[3] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Symposium on Principals & Practice of Parallel Programming*, July 1995.

[4] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[5] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *25nd Annual International Symposium on Computer Architecture*, July 1998.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[7] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Conference on Programming Language Design and Implementation*, June 2001.

[8] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications*, November 2002.

[9] B. Bigler, S. Allan, and R. Oldehoeft. Parallel dynamic storage allocation. In *International Conference on Parallel Processing*, 1985.

[10] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. Optimal strategies for spinning and blocking. Technical report, CSRI-278, Computer System Research Institute, University of Toronto, January 1993.

[11] J. Bradford and S. Abraham. Efficient synchronization for multithreaded processors. In *Workshop on Multithreaded Execution Architecture and Compilation*, January 1998.

[12] Compaq Computer Corporation. malloc(3) man page, Standard C library for Tru64 UNIX.

[13] Z. Cvetanovic and R. Kessler. Performance analysis of the Alpha 21264-based Compaq ES40 System. In *27th International Symposium on Computer Architecture*, June 2000.

[14] Intel Developer. Introduction to next generation multiprocessing: Hyper-threading technology. http://developer.intel.com/technology/hyperthread/intro_nexgen/, August 2001.

[15] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5), August 1997.

[16] S. Eggers and R. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[17] D. Haggander and L. Lundberg. Attacking the dynamic memory problem for SMPs. In *13th International Conference on Parallel and Distributed Computing System*, 2000.

[18] htdig. WWW search engine software. www.htdig.org.

[19] A. Iyengar. Parallel dynamic storage allocation algorithms. In *Fifth IEEE Symposium on Parallel and Distributed Processing*, 1993.

[20] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Symposium on Principals & Practice of Parallel Programming*, July 1995.

[21] M. S. Johnstone and P. R. Wilson. *The memory fragmentation problem: Solved?*. In *ISSM*, Vancouver, B.C., Canada, 1998.

[22] A. Kagi, D. Burger, and J. R. Goodman. Efficient synchronization: Let them eat QOLB. In *24nd International Symposium on Computer Architecture*, June 1997.

[23] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principle*, 1991.

[24] P. Kerly. Adjusting thread stack address to improve performance on intel xeon processors. Intel Developer Services. http://cedar.intel.com/cgi-bin/ids.dll/topic.jsp?catCode=CYU, February 2002.

[25] H. Kwak, B. Lee, A. R. Hurson, S.-H. Yoon, and W.-J. Hahn. Effects of multithreading on cache performance. *IEEE Transactions on Computers*, 48(2), 1999.

[26] P.-. Larson and M. Krishnan. Memory allocation for long-running server applications. In *International Symposium on Memory Management*, 1998.

[27] J. R. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *USENIX Annual Technical Conference*, 2002.

[28] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[29] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, J. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreading processors. In *25nd Annual International Symposium on Computer Architecture*, June 1998.

[30] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(2), August 1997.

[31] J. Lo, S. Eggers, H. Levy, S. Parekh, and D. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[32] L. McDowell. Exploring the performance potential of a staged software architecture on simultaneous multithreaded processors. In *Qualifying Examination Report, University of Washington*, July 2001.

[33] S. McFarling. Combining branch predictors. Technical report, TN-36, DEC-WRL, June 1993.

[34] S. Microsystems. A comparison of memory allocators in multiprocessors. http://soldc.sun.com/articles/multiproc/multiproc.html.

[35] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX 1999 Annual Technical Conference*, 1999.

[36] R. Preston et al. Design of an 8-wide superscalar risc microprocessor with simultaneous multithreading. In *International Solid-State Circuits Conference*, February 2002.

[37] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th International Symposium on Microarchitecture*, December 2001.

[38] J. Redstone. *An Analysis of Software Interface Issues for SMT Processors*. Ph.D. thesis, University of Washington, December 2002.

[39] J. Redstone, S. Eggers, and H. Levy. Analysis of operating system behavior on a simultaneous multithreaded architecture. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[40] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[41] J.-Y. Tsai, Z. Jiang, E. Ness, and P.-C. Yew. Performance study of a concurrent multithreaded processor. In *Fourth International Symposium on High Performance Computer Architecture*, February 1998.

[42] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23nd Annual International Symposium on Computer Architecture*, May 1996.

[43] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[44] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting fine-grain synchronization on a simultaneous multithreaded processor. In *Fifth International Conference on High-Performance Computer Architecture*, January 1999.

[45] V.-Y. Vee and W.-J. Hsu. A scalable and efficient storage allocator on shared-memory multiprocessors. *Parallel Processing Letters*, 11(2/3), 2001.

[46] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, 2001.

[47] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, June 1995.