Capriccio: Scalable Threads for Internet Services

Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer
Computer Science Division
University of California, Berkeley
{jrvb,jcondit,zf,necula,brewer}@cs.berkeley.edu

ABSTRACT

This paper presents Capriccio, a scalable thread package for use with high-concurrency servers. While recent work has advocated event-based systems, we believe that thread-based systems can provide a simpler programming model that achieves equivalent or superior performance.

By implementing Capriccio as a user-level thread package, we have decoupled the thread package implementation from the underlying operating system. As a result, we can take advantage of cooperative threading, new asynchronous I/O mechanisms, and compiler support. Using this approach, we are able to provide three key features: (1) scalability to 100,000 threads, (2) efficient stack management, and (3) resource-aware scheduling.

We introduce *linked stack management*, which minimizes the amount of wasted stack space by providing safe, small, and non-contiguous stacks that can grow or shrink at run time. A compiler analysis makes our stack implementation efficient and sound. We also present *resource-aware scheduling*, which allows thread scheduling and admission control to adapt to the system's current resource usage. This technique uses a *blocking graph* that is automatically derived from the application to describe the flow of control between blocking points in a cooperative thread package. We have applied our techniques to the Apache 2.0.44 web server, demonstrating that we can achieve high performance and scalability despite using a simple threaded programming model.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*threads*

General Terms

Algorithms, Design, Performance

Keywords

user-level threads, linked stack management, dynamic stack growth, resource-aware scheduling, blocking graph

1. INTRODUCTION

Today's Internet services have ever-increasing scalability demands. Modern servers must be capable of handling tens or hundreds of thousands of simultaneous connections without significant performance degradation. Current commodity hardware is capable of meeting these demands, but software has lagged behind. In particular, there is a pressing need for a programming model that allows programmers to design efficient and robust servers with ease.

Thread packages provide a natural abstraction for high-concurrency programming, but in recent years, they have been supplanted by event-based systems such as SEDA [41]. These event-based systems handle requests using a pipeline of stages. Each request is represented by an event, and each stage is implemented as an event handler. These systems allow precise control over batch processing, state management, and admission control; in addition, they provide benefits such as atomicity within each event handler.

Unfortunately, event-based programming has a number of drawbacks when compared to threaded programming [39]. Event systems hide the control flow through an application, making it difficult to understand cause and effect relationships when examining source code and when debugging. For instance, many event systems invoke a method in another module by sending a "call" event and then waiting for a "return" event in response. In order to understand the application, the programmer must mentally match these call/return pairs, even when they are in different parts of the code. Furthermore, creating these call/return pairs often requires the programmer to manually save and restore live state. This process, referred to as "stack ripping" [1], is a major burden for programmers who wish to use event systems.

In this paper, we advocate a different solution: instead of switching to an event-based model to achieve high concurrency, we should fix the thread-based model. We believe that a modern thread package will be able to provide the same benefits as an event system while also offering a better programming model for Internet services. Specifically, our goals for our revised thread package are:

- Support for existing thread APIs.
- Scalability to hundreds of thousands of threads.
- Flexibility to address application-specific needs.

In meeting these goals, we have made it possible for programmers to write high-performance Internet servers using the intuitive one-thread-per-connection programming style.

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

Indeed, our thread package can improve performance of existing threaded applications with little to no modification of the application itself.

1.1 Thread Design Principles

In the process of "fixing" threads for use in server applications, we found that a user-level approach is essential. While user-level threads and kernel threads are both useful, they solve fundamentally different problems. Kernel threads are primarily useful for enabling true concurrency via multiple devices, disk requests, or CPUs. User-level threads are really logical threads that should provide a clean programming model with useful invariants and semantics.

To date, we do not strongly advocate any *particular* semantics for threads; rather, we argue that any clean semantics for threads requires *decoupling* the threads of the programming model (logical threads) from those of the underlying kernel. Decoupling the programming model from the kernel is important for two reasons. First, there is substantial variation in interfaces and semantics among modern kernels, despite the existence of the POSIX standard. Second, kernel threads and asynchronous I/O interfaces are areas of active research [22, 23]. The range of semantics and the rate of evolution both require decoupling: logical threads can hide both OS variation and kernel evolution.

In our case, this decoupling has provided a number of advantages. We have been able to integrate compiler support into our thread package, and we have taken advantage of several new kernel features. Thus, we have been able to increase performance, improve scalability, and address application-specific needs, all without changing application code.

1.2 Capriccio

This paper discusses our new thread package, Capriccio. This thread package achieves our goals with the help of three key features:

First, we improved the scalability of basic thread operations. We accomplished this task by using user-level threads with cooperative scheduling, by taking advantage of a new asynchronous I/O interface, and by engineering our runtime system so that all thread operations are $O(1)$.

Second, we introduced *linked stacks*, a mechanism for dynamic stack growth that solves the problem of stack allocation for large numbers of threads. Traditional thread systems preallocate large chunks of memory for each thread's stack, which severely limits scalability. Capriccio uses a combination of compile-time analysis and run-time checks to limit the amount of wasted stack space in an efficient and application-specific manner.

Finally, we designed a *resource-aware scheduler*, which extracts information about the flow of control within a program in order to make scheduling decisions based on predicted resource usage. This scheduling technique takes advantage of compiler support and cooperative threading to address application-specific needs without requiring the programmer to modify the original program.

The remainder of this paper discusses each of these three features in detail. Then, we present an overall experimental evaluation of our thread package. Finally, we discuss future directions for user-level thread packages with integrated compiler support.

2. THREAD DESIGN AND SCALABILITY

Capriccio is a fast, user-level thread package that supports the POSIX API for thread management and synchronization. In this section, we discuss the overall design of our thread package, and we demonstrate that it satisfies our scalability goals.

2.1 User-Level Threads

One of the first issues we explored when designing Capriccio was whether to employ user-level threads or kernel threads. User-level threads have some important advantages for both performance and flexibility. Unfortunately, they also complicate preemption and can interact badly with the kernel scheduler. Ultimately, we decided that the advantages of user-level threads are significant enough to warrant the additional engineering required to circumvent their drawbacks.

2.1.1 Flexibility

User-level threads provide a tremendous amount of flexibility for system designers by creating a level of indirection between applications and the kernel. This abstraction helps to decouple the two, and it allows faster innovation on both sides. For example, Capriccio is capable of taking advantage of the new asynchronous I/O mechanisms the development-series Linux kernel, which allows us to provide performance improvements without changing application code.

The use of user-level threads also increases the flexibility of the thread scheduler. Kernel-level thread scheduling must be general enough to provide a reasonable level of quality for all applications. Thus, kernel threads cannot tailor the scheduling algorithm to fit a specific application. Fortunately, user-level threads do not suffer from this limitation. Instead, the user-level thread scheduler can be built along with the application.

User-level threads are extremely lightweight, which allows programmers to use a tremendous number of threads without worrying about threading overhead. The benchmarks in Section 2.3 show that Capriccio can scale to $100,000$ threads; thus, Capriccio makes it possible to write highly concurrent applications (which are often written with messy, event-driven code) in a simple threaded style.

2.1.2 Performance

User-level threads can greatly reduce the overhead of thread synchronization. In the simplest case of cooperative scheduling on a single CPU, synchronization is nearly free, since neither user threads nor the thread scheduler can be interrupted while in a critical section.[1] In the future, we believe that flexible user-level scheduling and compile-time analysis will allow us to offer similar advantages on a multi-CPU machine.

Even in the case of preemptive threading, user-level threads offer an advantage in that they do not require kernel crossings for mutex acquisition or release. By comparison, kernel-level mutual exclusion requires a kernel crossing for every synchronization operation. While this situation can be improved for uncontended locks,[2] highly contended mutexes still require kernel crossings.

[1] Poorly designed signal handling code can reintroduce these problems, but this problem can easily be avoided.

[2] The `futexes` in recent Linux kernels allow operations on uncontended mutexes to occur entirely in user space.

Finally, memory management is more efficient with user-level threads. Kernel threads require data structures that eat up valuable kernel address space, decreasing the space available for I/O buffers, file descriptors, and other resources.

2.1.3 Disadvantages

User-level threading is not without its drawbacks, however. In order to retain control of the processor when a user-level thread executes a blocking I/O call, a user-level threading package overrides these blocking calls and replaces them internally with non-blocking equivalents. The semantics of these non-blocking I/O mechanisms generally require an increased number of kernel crossings when compared to the blocking equivalents. For example, the most efficient non-blocking network I/O primitive in Linux (`epoll`) involves first polling sockets for I/O readiness and then performing the actual I/O call. These second I/O calls are identical to those performed in the blocking case; the poll calls are additional overhead. Non-blocking disk I/O mechanisms are often similar in that they employ separate system calls to submit requests and retrieve responses.[3]

In addition, user-level thread packages must introduce a wrapper layer that translates blocking I/O mechanisms to non-blocking I/O ones, and this layer is another source of overhead. At best, this layer can be a very thin shim, which simply adds a few extra function calls. However, for quick operations such as in-cache reads that are easily satisfied by the kernel, this overhead can become important.

Finally, user-level threading can make it more difficult to take advantage of multiple processors. The performance advantage of lightweight synchronization is diminished when multiple processors are allowed, since synchronization is no longer "for free". Additionally, as discussed by Anderson et al. in their work on scheduler activations, purely user-level synchronization mechanisms are ineffective in the face of true concurrency and may lead to starvation [2].

Ultimately, we believe the benefits of user-level threading far outweigh these disadvantages. As the benchmarks in Section 2.3 show, the additional overhead incurred does not seem to be a problem in practice. In addition, we are working on ways to overcome the difficulties with multiple processors; we will discuss this issue further in Section 7.

2.2 Implementation

We have implemented Capriccio as a user-level threading library for Linux. Capriccio implements the POSIX threading API, which allows it to run most applications without modification.

Context Switches. Capriccio is built on top of Edgar Toernig's coroutine library [35]. This library provides extremely fast context switches for the common case in which threads voluntarily yield, either explicitly or through making a blocking I/O call. We are currently designing signal-based code that allows for preemption of long-running user threads, but Capriccio does not provide this feature yet.

I/O. Capriccio intercepts blocking I/O calls at the library level by overriding the system call stub functions in GNU libc. This approach works flawlessly for statically linked applications and for dynamically linked applications that use GNU libc versions 2.2 and earlier. However, GNU libc version 2.3 bypasses the system call stubs for many of its internal routines (such as `printf`), which causes problems for dynamically linked applications. We are working to allow Capriccio to function as a libc add-on in order to provide better integration with the latest versions of GNU libc.

Internally, Capriccio uses the latest Linux asynchronous I/O mechanisms—`epoll` for pollable file descriptors (e.g., sockets, pipes, and fifos) and Linux AIO for disk. If these mechanisms are not available, Capriccio falls back on the standard Unix `poll()` call for pollable descriptors and a pool of kernel threads for disk I/O. Users can select among the available I/O mechanisms by setting appropriate environment variables prior to starting their application.

Scheduling. Capriccio's main scheduling loop looks very much like an event-driven application, alternately running application threads and checking for new I/O completions. Note, though, that the scheduler hides this event-driven behavior from the programmer, who still uses the standard thread-based abstraction. Capriccio has a modular scheduling mechanism that allows the user to easily select between different schedulers at run time. This approach has also made it simple for us to develop several different schedulers, including a novel scheduler based on thread resource utilization. We discuss this feature in detail in Section 4.

Synchronization. Capriccio takes advantage of cooperative scheduling to improve synchronization. At present, Capriccio supports cooperative threading on single-CPU machines, in which case inter-thread synchronization primitives require only simple checks of a boolean locked/unlocked flag. For cases in which multiple kernel threads are involved, Capriccio employs either spin locks or optimistic concurrency control primitives, depending on which mechanism best fits the situation.

Efficiency. In developing Capriccio, we have taken great care to choose efficient algorithms and data structures. Consequently, all but one of Capriccio's thread management functions has a bounded worst-case running time, independent of the number of threads. The sole exception is the sleep queue, which currently uses a naive linked list implementation. While the literature contains a number of good algorithms for efficient sleep queues, our current implementation has not caused problems yet, so we have focused our development efforts on other aspects of the system.

2.3 Threading Microbenchmarks

We ran a number of microbenchmarks to validate Capriccio's design and implementation. Our test platform was an SMP with two 2.4 GHz Xeon processors, 1 GB of memory, two 10K RPM SCSI Ultra II hard drives, and 3 Gigabit Ethernet interfaces. The operating system was Linux 2.5.70, which includes support for `epoll`, asynchronous disk I/O, and lightweight system calls (`vsyscall`). We ran our benchmarks on three thread packages: Capriccio, LinuxThreads (the standard Linux kernel thread package), and NPTL version 0.53 (the new Native POSIX Threads for Linux

[3] Although there are non-blocking I/O mechanisms (such as POSIX AIO's `lio_listio()` and Linux's new `io_submit()`) that allow the submission of multiple I/O requests with a single system call, there are other issues that make this feature difficult to use. For example, implementations of POSIX AIO often suffer from performance problems. Additionally, use of batching creates a trade-off between system call overhead and I/O latency, which is difficult to manage.

	Capriccio	Capriccio_notrace	LinuxThreads	NPTL
Thread creation	21.5	21.5	37.9	17.7
Thread context switch	0.56	0.24	0.71	0.65
Uncontended mutex lock	0.04	0.04	0.14	0.15

Table 1: Latencies (in μs) of thread primitives for different thread packages.

package). We built all applications with gcc 3.3 and linked against GNU libc 2.3. We recompiled LinuxThreads to use the new lightweight system call feature of latest Linux kernels to ensure a fair comparison with NPTL, which uses this feature.

2.4 Thread Primitives

Table 1 compares average times of several thread primitives for Capriccio, LinuxThreads, and NPTL. In the test labeled `Capriccio_notrace`, we disabled statistics collection and dynamic stack backtracing (used for the scheduler discussed in Section 4) to show their impact on performance. Thread creation time is dominated by stack allocation time and is quite expensive for all four thread packages. Thread context switches, however, are significantly faster in Capriccio, even with the stack tracing and statistics collection overhead. We believe that reduced kernel crossings and our simpler scheduling policy both contributed to this result. Synchronization primitives are also much faster in Capriccio (by a factor of 4 for uncontended mutex locking) because no kernel crossings are involved.

2.5 Thread Scalability

To measure the overall efficiency and scalability of scheduling and synchronization in different thread packages, we ran a simple producer-consumer microbenchmark on the three packages. Producers put empty messages into a shared buffer, and consumers "process" each message by looping for a random amount of time. Synchronization is implemented using condition variables and mutexes. Equal numbers of producers and consumers are created for each test. Each test is run for 10 seconds and repeated 5 times. Average throughput and standard deviations are shown in Figure 1. Capriccio outperforms NPTL and LinuxThreads in terms of both raw performance and scalability. Throughput of LinuxThreads begins to degrade quickly after only 20 threads are created, and NPTL's throughput degrades after 100. NPTL shows unstable behavior with more than 64 threads, which persists across two NPTL versions (0.53 and 0.56) and several 2.5 series kernels we tested. Capriccio scales to 32K producers and consumers (64K threads total). We attribute the drop of throughput between 100 threads and 1000 to increased cache footprint.

2.6 I/O Performance

Figure 2 shows the network performance of Capriccio and other thread packages under load. In this test, we measured the throughput of concurrently passing a number of tokens (12 bytes each) among a fixed number of pipes. The number of concurrent tokens is one quarter of the number of pipes if there are less than 128 pipes; otherwise, there are exactly 128 tokens. The benchmark thus simulates the effect of slow client links—that is, a large number of mostly-idle pipes. This scenario is typical for Internet servers, and traditional threading systems often perform poorly in such tests. Two functionally equivalent benchmark programs are used to obtain the results: a threaded version is used for Capriccio, LinuxThreads, and NPTL, and a non-blocking I/O version is used for `poll` and `epoll`. Five million tokens are passed for each test and each test is run five times.

The figure shows that Capriccio scales smoothly to 64K threads and incurs less than 10% overhead when compared to `epoll` with more than 256 pipes. To our knowledge, `epoll` is the best non-blocking I/O mechanism available on Linux; hence, its performance should reflect that of the best event-based servers, which all rely on such a mechanism. Capriccio performs consistently better than Poll, LinuxThreads, and NPTL with more than 256 threads and is more than twice as fast as both LinuxThreads and NPTL when more than 1000 threads are created.

However, when concurrency is low (< 100 pipes), Capriccio is slower than its competitors because it issues more system calls. In particular, it calls `epoll_wait()` to obtain file descriptor readiness events to wake up threads blocking for I/O. It performs these calls periodically, transferring as many events as possible on each call. However, when concurrency is low, the number of runnable threads occasionally reaches zero, forcing Capriccio to issue more `epoll_wait()` calls. In the worst case, Capriccio is 37% slower than NPTL when there are only 2 concurrent tokens (and 8 threads). Fortunately, this overhead is amortized quickly when concurrency increases; more scalable scheduling allows Capriccio to outperform LinuxThreads and NPTL at high concurrency.

Since Capriccio uses asynchronous I/O primitives, Capriccio can benefit from the kernel's disk head scheduling algorithm just as much as kernel threads can. Figure 3 shows a microbenchmark in which a number of threads perform random 4 KB reads from a 1 GB file. The test program bypasses the kernel buffer cache by using `O_DIRECT` when opening the file. Each test is run for 10 seconds and averages of 5 runs are shown. Throughput of all three thread libraries increases steadily with the concurrency level until it levels off when concurrency reaches about 100. In contrast, utilization of the kernel's head scheduling algorithm in event-based systems that use blocking disk I/O (e.g., SEDA) is limited by the number of kernel threads used, which is often made deliberately small to reduce kernel scheduling overhead. Even worse, other process-based applications that use non-blocking I/O (either `poll()`, `select()`, `/dev/poll`, or `epoll`) cannot benefit from the kernel's head scheduling at all if they do not explicitly use asynchronous I/O. Unfortunately, most programs do not use asynchronous I/O because it significantly increases programming complexity and compromises portability.

Figure 4 shows disk I/O performance of the three thread libraries when using the OS buffer cache. In this test, we measure the throughput achieved when 200 threads read continuously 4K blocks from the file system with a specified buffer cache miss rate. The cache miss rate is fixed by reading an appropriate portion of data from a small file opened normally (hence all cache hits) and by reading the

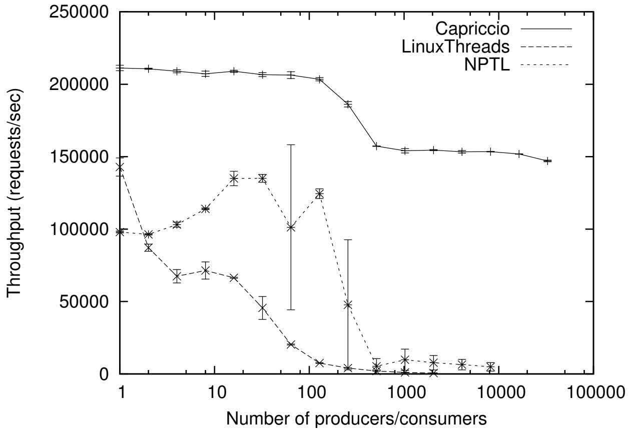

Figure 1: `Producer-Consumer` - scheduling and synchronization performance.

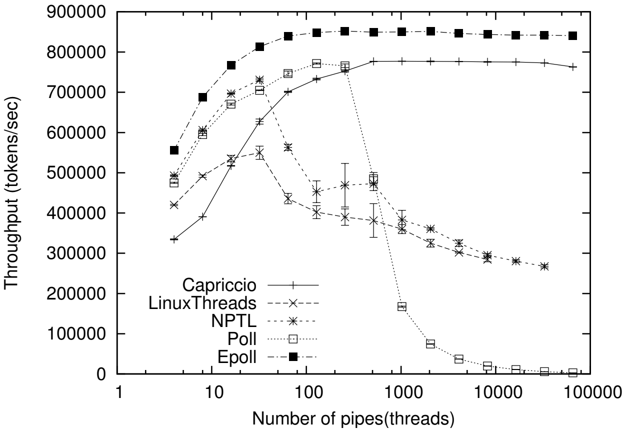

Figure 2: `Pipetest` - network scalability test.

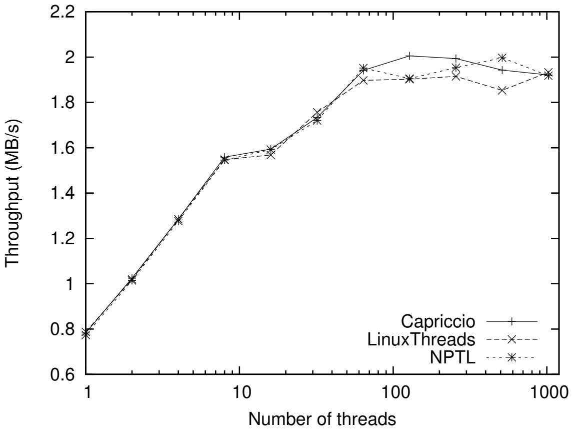

Figure 3: Benefits of disk head scheduling.

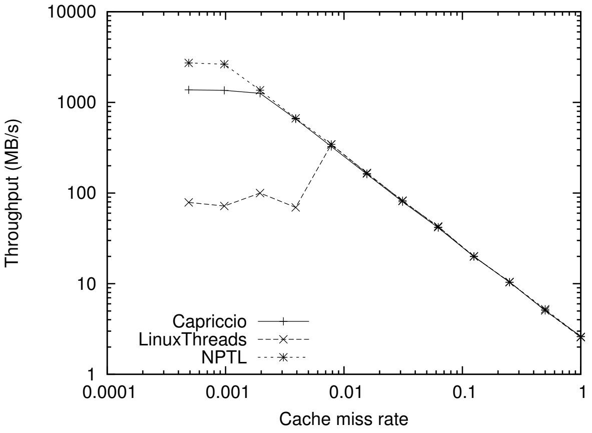

Figure 4: Disk I/O performance with buffer cache.

remaining data from a file opened with `O_DIRECT`. For a higher miss rate, the test is disk-bound; thus, Capriccio's performance is identical to that of NPTL and LinuxThreads. However, when the miss rate is very low, the program is CPU-bound, so throughput is limited by per-transfer overhead. Here, Capriccio's maximum throughput is about 50% of NPTL's, which means Capriccio's overhead is twice that of NPTL. The source of this overhead is the asynchronous I/O interface (Linux AIO) used by Capriccio, which incurs the same amount of overhead for cache-hitting operations and for ones that reach the disk: for each I/O request, a completion event needs to be constructed, queued, and delivered to user-level through a separate system call. However, this shortcoming is relatively easy to fix: by returning the result immediately for requests that do not need to wait, we can eliminate most (if not all) of this overhead. We leave this modification as future work. Finally, LinuxThreads' performance degrades significantly at a very low miss rate. We believe this degradation is a result of a bug either in the kernel or in the library, since the processor is mostly idle during the test.

3. LINKED STACK MANAGEMENT

Thread packages usually attempt to provide the programmer with the abstraction of an unbounded call stack for each thread. In reality, the stack size is bounded, but the bounds are chosen conservatively so that there is plenty of space for normal program execution. For example, LinuxThreads allocates two megabytes per stack by default; with such a conservative allocation scheme, we consume 1 GB of virtual memory for stack space with just 500 threads. Fortunately, most threads consume only a few kilobytes of stack space at any given time, although they might go through stages when they use considerably more. This observation suggests that we can significantly reduce the size of virtual memory dedicated to stacks if we adopt a dynamic stack allocation policy wherein stack space is allocated to threads on demand in relatively small increments and is deallocated when the thread requires less stack space. In the rest of this section, we discuss a compiler feature that allows us to provide such a mechanism while preserving the programming abstraction of unbounded stacks.

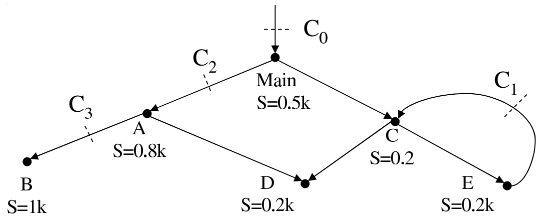

Figure 5: An example of a call graph annotated with stack frame sizes. The edges marked with C_i (i=$0, \ldots, 3$) are the checkpoints.

3.1 Compiler Analysis and Linked Stacks

Our approach uses a compiler analysis to limit the amount of stack space that must be preallocated. We perform a whole-program analysis based on a *weighted call graph*.[4] Each function in the program is represented by a node in this call graph, weighted by the maximum amount of stack space that a single stack frame for that function will consume. An edge between node A and node B indicates that function A calls function B directly. Thus, paths between nodes in this graph correspond to sequences of stack frames that may appear on the stack at run time. The length of a path is the sum of the weights of all nodes in this path; that is, it is the total size of the corresponding sequence of stack frames. An example of such a graph is shown in Figure 5.

Using this call graph, we wish to place a reasonable bound on the amount of stack space that will be consumed by each thread. If there are no recursive functions in our program, there will be no cycles in the call graph, and thus we can easily bound the maximum stack size for the program at compile time by finding the longest path starting from each thread's entry point. However, most real-world programs make use of recursion, which means that we cannot compute a bound on the stack size at compile time. And even in the absence of recursion, the static computation of stack size might be too conservative. For example, consider the call graph in Figure 5. Ignoring the cycle in the graph, the maximum stack size is 2.3 KB on the path `Main`–`A`–`B`. However, the path `Main`–`C`–`D` has a smaller stack size of only 0.9 KB. If the first path is only used during initialization and the second path is used through the program's execution, then allocating 2.3 KB to each thread would be wasteful. For these reasons, it is important to be able to grow and shrink the stack size on demand.

In order to implement dynamically-sized stacks, our call graph analysis identifies call sites at which we must insert *checkpoints*. A checkpoint is a small piece of code that determines whether there is enough stack space left to reach the next checkpoint without causing stack overflow. If not enough space remains, a new *stack chunk* is allocated, and the stack pointer is adjusted to point to this new chunk. When the function call returns, the stack chunk is unlinked and returned to a free list.

This scheme results in non-contiguous stacks, but because the stack chunks are switched right before the actual arguments for a function call are pushed, the code for the callee need not be changed. And because the caller's frame pointer is stored on the callee's stack frame, debuggers can follow the backtrace of a program.[5] The code for a checkpoint is written in C, with a small amount of inline assembly for reading and setting of the stack pointer; this code is inserted using a source-to-source transformation of the program prior to compilation. Mutual exclusion for accessing the free stack chunk list is ensured by our cooperative threading approach.

3.2 Placing Checkpoints

During our program analysis, we must determine where to place checkpoints. A simple solution is to insert checkpoints at every call site; however, this approach is prohibitively expensive. A less restrictive approach is to ensure that at each checkpoint, we have a bound on the stack space that may be consumed before we reach the next checkpoint (or a leaf in the call graph).

To satisfy this requirement, we must ensure that there is at least one checkpoint in every cycle within the call graph (recall that the edges in the call graph correspond to call sites). To find the appropriate points to insert checkpoints, we perform a depth-first search on the call graph, which identifies back edges—that is, edges that connect a node to one of its ancestors in the call graph [25]. All cycles in the graph must contain a back edge, so we add checkpoints at all call sites identified as back edges in order to ensure that any path from a function to a checkpoint has bounded length. In Figure 5, the checkpoint C_0 allocates the first stack chunk, and the checkpoint C_1 is inserted on the back edge `E`–`C`.

Even after we break all cycles, the bounds on stack size may be too large. Thus, we add additional checkpoints to the graph to ensure that all paths between checkpoints are within a desired bound, which is given as a compile-time parameter. To insert these new checkpoints, we process the call graph once more, this time determining the longest path from each node to the next checkpoint or leaf. When performing this analysis, we consider a restricted call graph that does not contain any back edges, since these edges already have checkpoints. This restricted graph has no cycles, so we can process the nodes bottom-up; thus, when processing node n, we will have already determined the longest path for each of n's successors. So, for each successor s of node n, we take the longest path for s and add n. If this new path's length exceeds the specified path limit parameter, we add a checkpoint to the edge between n and s, which effectively reduces the longest path of s to zero. The result of this algorithm is a set of edges where checkpoints should be added along with reasonable bounds on the maximum path length from each node. For the example in Figure 5, with a limit of 1 KB, this algorithm places the additional checkpoints C_2 and C_3. Without the checkpoint C_2, the stack frames of `Main` and `A` would use more than 1 KB.

Figure 6 shows four instances in the lifetime of the thread whose call graph is shown in Figure 5. In Figure 6(a), the function B is executing, with three stack chunks allocated at checkpoints C_0, C_2, and C_3. Notice that 0.5 KB is wasted in the first stack chunk, and 0.2 KB is wasted in the second

[4] We use the CIL toolkit [26] for this purpose, which allows efficient whole-program analysis of real-world applications like the Apache web server.

[5] This scheme does not work when the `omit-frame-pointer` is enabled in `gcc`. It is possible to support this optimization by using more expensive checkpoint operations such as copying the arguments from the caller's frame to the callee's frame.

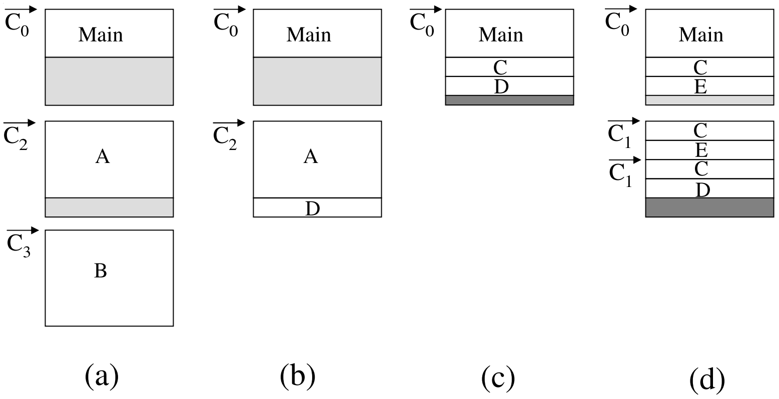

Figure 6: Examples of dynamic allocation and deallocation of stack chunks.

chunk. In Figure 6(b), function A has called D, and only two stack chunks were necessary. Finally, in Figure 6(d) we see an instance with recursion. A new stack chunk is allocated when E calls C (at checkpoint C_1). However, the second time around, the code at checkpoint C_1 decides that there is enough space remaining in the current stack chunk to reach either a leaf function (D) or the next checkpoint (C_1).

3.3 Dealing with Special Cases

Function pointers present an additional challenge to our algorithm, because we do not know at compile time exactly which function may be called through a given function pointer. To improve the results of our analysis, though, we want to determine as precisely as possible the set of functions that might be called at a function pointer call site. Currently, we categorize function pointers by number and type of arguments, but in the future, we plan to use a more sophisticated pointer analysis.

Calls to external functions also cause problems, since it is more difficult to bound the stack space used by precompiled libraries. We provide two solutions to this problem. First, we allow the programmer to annotate external library functions with trusted stack bounds. Alternatively, we allow larger stack chunks to be linked for external functions; as long as threads don't block frequently within these functions, we can reuse a small number of large stack chunks throughout the application. For the C standard library, we use annotations to deal with functions that block or functions that are frequently called; these annotations were derived by analyzing library code.

3.4 Tuning the Algorithm

Our algorithm causes stack space to be wasted in two places. First, some stack space is wasted when a new stack chunk is linked; we call this space *internal wasted space*. Second, stack space at the bottom of the current chunk is considered unused; this space is called *external wasted space*. In Figure 6, internal wasted space is shown in light gray, whereas external wasted space is shown in dark gray.

The user is allowed to tune two parameters that adjust the trade-offs in terms of wasted space and execution speed. First, the user can adjust *MaxPath*, which specifies the maximum desired path length in the algorithm we have just described. This parameter affects the trade-off between execution time and internal wasted space; larger path lengths require fewer checkpoints but more stack linking. Second, the user can adjust *MinChunk*, the minimum stack chunk size. This parameter affects the trade-off between stack linking and external wasted space; larger chunks result in more external wasted space but less frequent stack linking, which in turn results in less internal wasted space and a smaller execution time overhead. Overall, these parameters provide a useful mechanism allowing the user (or the compiler) to optimize memory usage.

3.5 Memory Benefits

Our linked stack technique has a number of advantages in terms of memory performance. In general, these benefits are achieved by divorcing thread implementation from kernel mechanisms, thus improving our ability to tune individual application memory usage. Compiler techniques make this application-specific tuning practical.

First, our technique makes preallocation of large stacks unnecessary, which in turn reduces virtual memory pressure when running large numbers of threads. Our analysis achieves this goal without the use of guard pages, which would contribute unnecessary kernel crossings and virtual memory waste.

Second, using linked stacks can improve paging behavior significantly. Linked stack chunks are reused in LIFO order, which allows stack chunks to be shared between threads, reducing the size of the application's working set. Also, we can allocate stack chunks that are smaller than a single page, thus reducing the overall amount of memory waste.

To demonstrate the benefit of our approach with respect to paging, we created a microbenchmark in which each thread repeatedly calls a function `bigstack()`, which touches all pages of a 1 MB buffer on the stack. Threads yield between calls to `bigstack()`. Our compiler analysis inserts a checkpoint at these calls, and the checkpoint causes a large stack chunk to be linked only for the duration of the call. Since `bigstack()` does not yield, all threads share a single 1 MB stack chunk; without our stack analysis, we would have to give each thread its own individual 1 MB stack.

We ran this microbenchmark with 800 threads, each of which calls `bigstack()` 10 times. We recorded execution time for five runs of the test and averaged the results. When each thread has its own individual stack, the benchmark takes 3.33 seconds, 1.07 seconds of which are at user level. When using our stack analysis, the benchmark takes 1.04

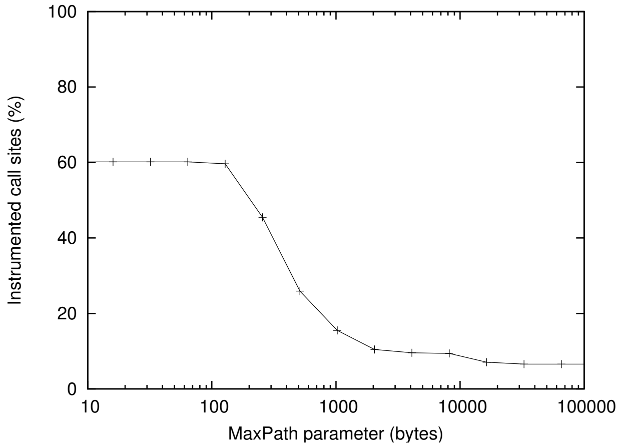

Figure 7: Number of Apache 2.0.44 call sites instrumented as a function of the *MaxPath* parameter.

seconds, with 1.00 seconds at user level. All standard deviations were within 0.02 seconds. The fact that total exeuction time decreases by a factor of three while user-level execution time remains roughly the same suggests that sharing a single stack via our linked stack mechanism drastically reduces the cost of paging. When running this test with 1,000 threads, the version without our stack analysis starts thrashing; with the stack analysis, though, the running time scales linearly up to 100,000 threads.

3.6 Case Study: Apache 2.0.44

We applied this analysis to the Apache 2.0.44 web server. We set the *MaxPath* parameter to 2 KB; this choice was made by examining the number of call sites instrumented for various parameter values. The results, shown in Figure 7, indicate that 2 KB or 4 KB is a reasonable choice, since larger parameter values make little difference in the overall amount of instrumentation. We set the *MinChunk* parameter to 4 KB based on profiling information. By adding profiling counters to checkpoints, we determined that increasing the chunk size to 4 KB reduced the number of stack links and unlinks significantly, but further increases yielded no additional benefit. We expect that this tuning methodology can be automated as long as the programmer supplies a reasonable profiling workload.

Using these parameters, we studied the behavior of Apache during execution of a workload consisting of static web pages based on the SPECweb99 benchmark suite. We used the threaded client program from the SEDA work [41] with 1000 simulated clients, a 10ms request delay, and a total file workload of 32 MB. The server ran 200 threads, using standard Unix `poll()` for network I/O and blocking for disk I/O. The total virtual memory footprint for Apache was approximately 22 MB, with a resident set size of approximately 10 MB. During this test, most functions could be executed entirely within the initial 4 KB chunk; when necessary, though, threads linked a 16 KB chunk in order to call a function that has an 8 KB buffer on its stack. Over five runs of this benchmark, the maximum number of 16 KB chunks needed at any given time had a mean of 66 (standard deviation 4.4). Thus, we required just under 8 MB of stack space overall: 800 KB for the initial stacks, 1 MB for larger chunks, and 6 MB for three 2 MB chunks used to run external functions. However, we believe that additional 16 KB chunks will be needed when using high-performance I/O mechanisms; we are still in the process of studying the impact of these features on stack usage. And while using an average of 66 16 KB buffers rather than one for each of the 200 threads is clearly a win, the addition of internal and external wasted space makes it difficult to directly compare our stack utilization with that of unmodified Apache. Nevertheless, this example shows that we are capable of running unmodified applications with a small amount of stack space without fear of stack overflow. Indeed, it is important to note that we provide safety in addition to efficiency; even though the unmodified version of Apache could run this workload with a single, contiguous 20 KB stack, this setting may not be safe for other workloads or for different configurations of Apache.

We observed the program's behavior at each call site crossed during the execution of this benchmark. The results were extremely consistent across five repetitions of the benchmark; thus, the numbers below represent the entire range of results over all five repetitions. At 0.1% of call sites, checkpoints caused a new stack chunk to be linked, at a cost of 27 instructions. At 0.4–0.5% of call sites, a large stack chunk was linked unconditionally in order to handle an external function, costing 20 instructions. At 10% of call sites, a checkpoint determined that a new chunk was not required, which cost 6 instructions. The remaining 89% of call sites were unaffected. Assuming all instructions are roughly equal in cost, the result is a 71–73% slowdown when considering function calls alone. Since call instructions make up only 5% of the program's instructions, the overall slowdown is approximately 3% to 4%.

4. RESOURCE-AWARE SCHEDULING

One of the advantages claimed for event systems is that their scheduling can easily adapt to the application's needs. Event-based applications are broken into distinct event handlers, and computation for a particular task proceeds as that task is passed from handler to handler. This architecture provides two pieces of information that are useful for scheduling. First, the current handler for a task provides information about the task's location in the processing chain. This information can be used to give priority to tasks that are closer to completion, hence reducing load on the system. Second, the lengths of the handlers' task queues can be used to determine which stages are bottlenecks and can indicate when the server is overloaded.

Capriccio provides similar application-specific scheduling for thread-based applications. Since Capriccio uses a cooperative threading model, we can view an application as a sequence of stages, where the stages are separated by blocking points. In this sense, Capriccio's scheduler is quite similar to an event-based system's scheduler. Our methods are more powerful, however, in that they deduce the stages automatically and have direct knowledge of the resources used by each stage, thus enabling finer-grained dynamic scheduling decisions. In particular, we use this automated scheduling to provide admission control and to improve response time.

Our approach allows Capriccio to provide sophisticated, application-specific scheduling without requiring the programmer to use complex or brittle tuning APIs. Thus, we

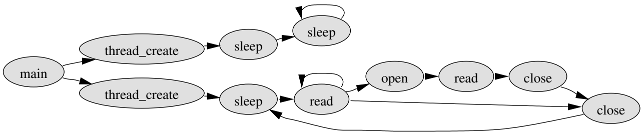

Figure 8: An example blocking graph. This graph was generated from a run of Knot, our test web server.

can improve performance and scalability without compromising the simplicity of the threaded programming model.

4.1 Blocking Graph

The key abstraction we use for scheduling is the *blocking graph*, which contains information about the places in the program where threads block. Each node is a location in the program that blocked, and an edge exists between two nodes if they were consecutive blocking points. The "location" in the program is not merely the value of the program counter, but rather the call chain that was used to reach the blocking point. This path-based approach allows us to differentiate blocking points in a more useful way than the program counter alone would allow, since otherwise there tend to be very few such points (e.g., the `read` and `write` system calls). Figure 8 shows the blocking graph for Knot, a simple thread-based web server. Each thread walks this graph independently, and every blocked thread is located at one of these nodes.

Capriccio generates this graph at run time by observing the transitions between blocking points. The key idea behind this approach is that Capriccio can *learn* the behavior of the application dynamically and then use that information to improve scheduling and admission control. This technique works in part because we are targeting long-running programs such as Internet servers, so it is acceptable to spend time learning in order to make improved decisions later on.

To make use of this graph when scheduling threads, we must annotate the edges and nodes with information about thread behavior. The first annotation we introduce is the average running time for each edge. When a thread blocks, we know which edge was just traversed, since we know the previous node. We measure the time it took to traverse the edge using the cycle counter, and we update an exponentially weighted average for that edge.

We keep a similar weighted average for each node, which we update every time a thread traverses one of its outgoing edges. Each node's average is essentially a weighted average of the edge values, since the number of updates is proportional to the number of times each outgoing edge is taken. The node value thus tells us how long the next edge will take *on average*.

Finally, we annotate the changes in resource usage. Currently, we define resources as memory, stack space, and sockets, and we track them individually. As with CPU time, there are weighted averages for both edges and nodes. Given that a blocked thread is located at a particular node, these annotations allows us to estimate whether running this thread will increase or decrease the thread's usage of each resource. This estimate is the basis for resource-aware scheduling: once we know that a resource is scarce, we promote nodes (and thus threads) that release that resource and demote nodes that acquire that resource.

4.2 Resource-Aware Scheduling

Most existing event systems prioritize event handlers statically. SEDA uses information such as event handler queue lengths to dynamically tune the system. Capriccio goes one step further by introducing the notion of resource-aware scheduling. In this section, we show how to use the blocking graph to perform resource-aware scheduling that is both transparent and application-specific.

Our strategy for resource-aware scheduling has three parts:

1. Keep track of resource utilization levels and decide dynamically if each resource is at its limit.

2. Annotate each node with the resources used on its outgoing edges so we can predict the impact on each resource should we schedule threads from that node.

3. Dynamically prioritize nodes (and thus threads) for scheduling based on information from the first two parts.

For each resource, we increase utilization until it reaches maximum capacity (so long as we don't overload another resource), and then we throttle back by scheduling nodes that release that resource. When resource usage is low, we want to preferentially schedule nodes that consume that resource, under the assumption that doing so will increase throughput. More importantly, when a resource is overbooked, we preferentially schedule nodes that release the resource to avoid thrashing.

This combination, when used with some hysteresis, tends to keep the system at full throttle without the risk of thrashing. Additionally, resource-aware scheduling provides a natural, workload-sensitive form of admission control, since tasks near completion tend to release resources, whereas new tasks allocate them. This strategy is completely adaptive, in that the scheduler responds to changes resource consumption due to both the type of work being done and offered load. The speed of adaptation is controlled by the parameters of the exponentially weighted averages in our blocking graph annotations.

Our implementation of resource-aware scheduling is quite straightforward. We maintain separate run queues for each node in the blocking graph. We periodically determine the relative priorities of each node based on our prediction of their subsequent resource needs and the overall resource utilization of the system. Once the priorities are known, we select a nodes by stride scheduling, and then we select threads within nodes by dequeuing from the nodes' run queues. Both of these operations are $O(1)$.

A key underlying assumption of our resource-aware scheduler is that resource usage is likely to be similar for many tasks at a blocking point. Fortunately, this assumption seems to hold in practice. With Apache, for example, there is almost no variation in resource utilization along the edges of the blocking graph.

4.2.1 Resources

The resources we currently track are CPU, memory, and file descriptors. We track memory usage by providing our own version of the `malloc()` family. We detect the resource limit for memory by watching page fault activity.

For file descriptors, we track the `open()` and `close()` calls. This technique allows us to detect an increase in open file descriptors, which we view as a resource. Currently, we set the resource limit by estimating the number of open connections at which response time jumps up.

We can also track virtual memory usage and number of threads, but we do not do so at present. VM is tracked the same way as physical memory, but the limit is reached when we reach some absolute threshold for total VM allocated (e.g., 90% of the full address space).

4.2.2 Pitfalls

We encountered some interesting pitfalls when implementing Capriccio's resource-aware scheduler. First, determining the maximum capacity of a particular resource can be tricky. The utilization level at which thrashing occurs often depends on the workload. For example, the disk subsystem can sustain far more requests per second if the requests are sequential instead of random. Additionally, resources can interact, as when the VM system trades spare disk bandwidth to free physical memory. The most effective solution we have found is to watch for early signs of thrashing (such as high page fault rates) and to use these signs to indicate maximum capacity.

Unfortunately, thrashing is not always an easy thing to detect, since it is characterized by a decrease in productive work and an increase in system overhead. While we can measure overhead, productivity is inherently an application-specific notion. At present, we attempt to guess at throughput, using measures like the number of threads created and destroyed and the number of files opened and closed. Although this approach seems sufficient for applications such as Apache, more complicated applications might benefit from a threading API that allows them to explicitly inform the runtime system about their current productivity.

Application-specific resources also present some challenges. For example, application-level memory management hides resource allocation and deallocation from the runtime system. Additionally, applications may define other logical resources such as locks. Once again, providing an API through which the application can inform the runtime system about its logical resources may be a reasonable solution. For simple cases like memory allocators, it may also be possible to achieve this goal with the help of the compiler.

4.3 Yield Profiling

One problem that arises with cooperative scheduling is that threads may not yield the processor, which can lead to unfairness or even starvation. These problems are mitigated to some extent by the fact that all of the threads are part of the same application and are therefore mutually trusting. Nonetheless, failure to yield is still a performance problem that matters.

Because we annotate the graph dynamically with the running time for each edge, it is trivial to find those edges that failed to yield: their running times are typically orders of magnitude larger than the average edge. Our implementation allows the system operator to see the full blocking graph including edge time frequencies and resources used, by sending a `USR2` signal to the running server process.

This tool is very valuable when porting legacy applications to Capriccio. For example, in porting Apache, we found many places that did not yield sufficiently often. This result is not surprising, since Apache expects to run with preemptive threads. For example, it turns out that the `close()` call, which closes a socket, can sometimes take 5ms even though the documentation insists that it returns immediately when nonblocking I/O is selected. To fix this problem, we insert additional yields in our system call library, before and after the actual call to `close()`. While this solution does not fix the problem in general, it does allow us to break the long edge into smaller pieces. A better solution (which we have not yet implemented) is to use multiple kernel threads for running user-level threads. This approach would allow the use of multiple processors, and it would hide latencies from occasional uncontrollable blocking operations such as `close()` calls or page fault handling.

5. EVALUATION

The microbenchmarks presented in Section 2.3 show that Capriccio has good I/O performance and excellent scalability. In this section, we evaluate Capriccio's performance more generally under a realistic web server workload. Real-world web workloads involve large numbers of potentially slow clients, which provide good tests of both Capriccio's scalability and scheduling. We discuss the overhead of Capriccio's resource-aware scheduler in this context, and then we discuss how this scheduler can achieve automatic admission control.

5.1 Web Server Performance

The server machine for our web benchmarks is a 4x500 MHz Pentium server with 2GB memory and a Intel e1000 Gigabit Ethernet card. The operating system is stock Linux 2.4.20. Unfortunately, we found that the development-series Linux kernel used in the microbenchmarks discussed earlier became unstable when placed under heavy load. Hence, this experiment does not take advantage of `epoll` or Linux AIO. Similarly, we were not able to compare Capriccio against NPTL for this workload. We leave these additional experiments for future work.

We generated client load with up to 16 similarly configured machines across a Gigabit switched network. Both Capriccio and Haboob perform non-blocking network I/O with the standard UNIX `poll()` system call and use a thread pool for disk I/O. Apache 2.0.44 (configured to use POSIX threads) uses a combination of spin-polling on individual file descriptors and standard blocking I/O calls.

The workload for this test consisted of requests for 3.2 GB of static file data with various file sizes. The request frequencies for each size and for each file were designed to match those of the SPECweb99 benchmark. The clients for this test repeatedly connect to the server and issue a series of five requests, separated by 20ms pauses. For each client load level we ran the test for 4 minutes and based our measurements on the middle two minutes.

We used the client program from the SEDA work [41] because this program was simpler to set up on our client machines and because it allowed us to disable the dynamic content tests, thus preventing external CGI programs from competing with the web server for resources.

We limited the cache sizes of Haboob and Knot to 200 MB in order to force a good deal of disk activity. We used a minimal configuration for Apache, disabling all dynamic modules and access permission checking. Hence, it performed essentially the same tasks as Haboob and Knot.

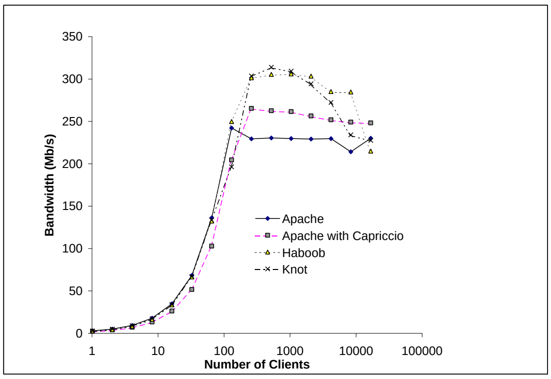

Figure 9: Web server bandwidth versus the number of simultaneous clients.

The performance results, shown in Figure 9 were quite encouraging. Apache's performance improved nearly 15% when run under Capriccio. Additionally, Knot's performance matched that of the event-based Haboob web server. While we do not have specific data on the variance of these results, it was quite small for load levels. There was more variation with more than 1024 clients, but the general trends were repeatable between runs.

Particularly remarkable is Knot's simplicity. Knot consists of 1290 lines of C code, written in a straightforward threaded style. Knot was very easy to write (it took one of us 3 days to create), and it is easy to understand. We consider this experience to be strong evidence for the simplicity of the threaded approach to writing highly concurrent applications.

5.2 Blocking Graph Statistics

Maintaining information about the resources used at each blocking point requires both determining where the program is when it blocks and performing some amount of computation to save and aggregate resource utilization figures.

Table 2 quantifies this overhead for Apache and Knot, for the workload described above. The top two lines show the average number of application cycles that each application spent going from one blocking point to the next. The bottom two lines show the number of cycles that Capriccio spends internally in order to maintain information used by the resource-aware scheduler. All cycle counts are the average number of cycles per blocking-graph edge during normal processing (i.e., under load and after the memory cache and branch predictors have warmed up).

It is important to note that these cycle counts include *only* the time spent in the application itself. Kernel time spent on I/O processing is not included. Since Internet applications are I/O intensive, much of their work actually takes place in the kernel. Hence, the performance impact of this overhead is lower than Table 2 would suggest.

The overhead of gathering and maintaining statistics is relatively small—less than 2% for edges in Apache. Moreover, these statistics tend to remain fairly steady in the workloads we have tested, so they can be sampled relatively infrequently. We have found a sampling ratio of 1/20 to be quite sufficient to maintain an accurate view of the system. This reduces the aggregate overhead to a mere 0.1%.

	Item	Cycles	Enabled
Apps	Apache	32697	n/a
	Knot	6868	n/a
System	stack trace	2447	Always for dynamic BG
	edge statistics	673	During sampling periods

Table 2: Average per-edge cycle counts for applications on Capriccio.

The overhead from stack traces is significantly higher, amounting to roughly 8% of the execution time for Apache and 36% for Knot. Additionally, since stack traces are essential for determining the location in the program, they must always be enabled.

The overhead from stack tracing illustrates how compiler integration could help to improve Capriccio's performance. The overhead to maintain location information in a statically generated blocking graph is essentially zero. Another more dynamic technique would be to maintain a global variable that holds a fingerprint of the current stack. This fingerprint can be updated at each function call by XOR'ing a unique function ID at each function's entry and exit point; these extra instructions can easily be inserted by the compiler. This fingerprint is not as accurate as a true stack trace, but it should be accurate enough to generate the same blocking graph that we currently use.

5.3 Resource-Aware Admission Control

To test our resource-aware admission control algorithms, we created a simple consumer-producer application. Producer threads loop, adding memory to a global pool and randomly touching pages to force them to stay in memory (or to cause VM faults for pages that have been swapped out). Consumer threads loop, removing memory from the global pool and freeing it.

This benchmark tests a number of system resources. First, if the producers allocate memory too quickly, the program may run out of virtual address space. Additionally, if page touching proceeds too quickly, the machine will thrash as the virtual memory system sends pages to and from disk. The goal, then, is to maximize the task throughput (measured by number of producer loops per second) while also making the best use of both memory and disk resources.

At run time, the test application is parameterized by the number of consumers and producers. Running under LinuxThreads, if there are more producers than consumers (and often when there are fewer) the system quickly starts to thrash. Under Capriccio, however, the resource-aware scheduler quickly detects the overload conditions and limits the number of producer threads from running. Thus, applications can reach a steady state near the knee of the performance curve.

6. RELATED WORK

Programming Models for High Concurrency

There has been a long-standing debate in the research community about the best programming model for high-concurrency; this debate has often focused on threads and events in particular. Ousterhout [28] enumerated a number of potential advantages for events. Similarly, recent work on scalable servers advocates the use of events. Examples include Internet servers such as Flash [29] and Harvest [10] and server infrastructures like SEDA [41] and Ninja [38].

In the tradition of the duality argument developed by Lauer and Needham [21], we have previously argued that any apparent advantages of events are simply artifacts of poor thread implementations [39]. Hence, we believe past arguments in favor of events are better viewed as arguments for application-specific optimization and the need for efficient thread runtimes. Both of these arguments are major motivations for Capriccio. Moreover, the blocking graph used by Capriccio's scheduler was directly inspired by SEDA's stages and explicit queues.

In previous work [39], we also presented a number of reasons that threads should be preferred over events for highly concurrent programming. This paper provides additional evidence for that claim by demonstrating Capriccio's performance, scalability, and ability to perform application-specific optimization.

Adya et al. [1] pointed out that the debate between event-driven and threaded programming can actually be split into two debates: one between preemptive and cooperative task management, and one between automatic and manual stack management. They coin the term "stack ripping" to describe the process of manually saving and restoring live state across blocking points, and they identify this process as the primary drawback to manual stack management. The authors also point out the advantages of the cooperative threading approach.

Many authors have attempted to improve threading performance by transforming threaded code to event based code. For example, Adya et al. [1] automate the process of "stack-ripping" in event-driven systems, allowing code to be written in a more thread-like style. In some sense, though, all thread packages perform this same translation at run time, by mapping blocking operations into non-blocking state machines underneath. Ultimately, we believe there is no advantage to a static transformation from threaded code to event-driven code, because a well-tuned thread runtime can perform just as well as an event-based one. Our performance tests with Capriccio corroborate this claim.

User-Level Threads

There have been many user-level thread packages, but they differ from Capriccio in their goals and techniques. To the best of our knowledge, Capriccio is unique in its use of the blocking graph to provide resource-aware scheduling and in its use of compile-time analysis to effect application-specific optimizations. Additionally, we are not aware of any language-independent threading library that uses linked stack frames, though we discuss some language-dependent ones below.

Filaments [31] and NT's Fibers are two high-performance user-level thread packages. Both use cooperative scheduling, but they are not targeted at large numbers of blocking threads. Minimal Context-Switching Threads [19] is a high-performance thread package specialized for web caches that includes fast disk libraries and memory management. The performance optimizations employed by these packages would be useful for Capriccio as well; these are complementary to our work.

The State Threads package [37] is a lightweight cooperative threading system that shares Capriccio's goal of simplifying the programming model for network servers. Unlike Capriccio, the State Threads library does not provide a POSIX threading interface, so applications must be rewritten to use it. Additionally, State Threads use either `select` or `poll` instead of the more scalable Linux `epoll`, and they use blocking disk I/O. These factors limit the scalability of State Threads for network-intensive workloads, and they restrict its concurrency for disk-intensive workloads. There are patches available to allow Apache to use State Threads [36], resulting in a performance increase. These patches include a number of other improvements to Apache, however, so it is impossible to tell how much of the improvement came from State Threads. Unfortunately, these patches are no longer maintained and do not compile cleanly, so we were unable to run direct comparisons against Capriccio.

Scheduler activations [2] solve the problem of blocking I/O and unexpected blocking/preemption of user-level threads by adding kernel support for notifying the user-level scheduler of these events. This approach ensures clean integration of the thread library and the operating system; however, the large amount of kernel changes involved seem to have precluded wide adoption. Another potential problem with this approach is that there will be one scheduler activation for each outstanding I/O operation, which can number in the tens of thousands for Internet servers. This result is contrary to the original goal of reducing the number of kernel threads needed. This problem apparently stems from the fact that scheduler activations are developed primarily for high performance computing environments, where disk and fast network I/O are dominant. Nevertheless, scheduler activations can be a viable approach to dealing with page faults and preemptions in Capriccio. Employing scheduler activations would also allow the user-level scheduler to influence the kernel's decision about which kernel thread to preempt. This scheme can be used to solve difficult problems like *priority inversion* and the *convoy phenomenon* [6].

Support for user-level preemption and M:N threading (i.e., running M user-level threads on top of N kernel threads) is tricky. Techniques such as optimistic concurrency control and Cilk's work-stealing [7] can be used effectively to manage thread and scheduler data structures. Cordina presents a nice description of these and other techniques in the context of Linux [12]. We expect to employ many of these techniques in Capriccio when we add support for M:N threading.

Kernel Threads

The NPTL project for Linux has made great strides toward improving the efficiency of Linux kernel threads. These advances include a number of kernel-level improvements such as better data structures, lower memory overhead, and the use of $O(1)$ thread management operations. NPTL is quite new and is still under active development. Hence, we expect that some of the performance degradation we found with higher numbers of threads may be resolved as the developers find bugs and create faster algorithms.

Application-Specific Optimization

Performance optimization through application-specific control of system resources is an important theme in OS research. Mach [24] allowed applications to specify their own VM paging scheme, which improved performance for applications that knew about their upcoming memory needs and disk access patterns. UNET [40] did similar things for network I/O, improving flexibility and reducing overhead without compromising safety. The SPIN operating system [5] and the VINO operating system [32] provide user customization by allowing application code to be moved into the kernel. The Exokernel [15] took the opposite approach and moved most of the OS to user level. All of these systems allow application-specific optimization of nearly all aspects of the system.

These techniques require programmers to tailor their application to manage resources for itself; this type of tuning is often difficult and brittle. Additionally, they tie programs to nonstandard APIs, reducing their portability. Capriccio takes a new approach to application-specific optimization by enabling automatic compiler-directed and feedback-based tuning of the thread package. We believe that this approach will make these techniques more practical and will allow a wider range of applications to benefit from them.

Asynchronous I/O

A number of authors propose improved kernel interfaces that could have an important impact on user-level threading. Asynchronous I/O primitives such as Linux's `epoll` [23], disk AIO [20] and FreeBSD's kqueue interface [22] are central to creating a scalable user-level thread package. Capriccio takes advantage of these interfaces and would benefit from improvements such as reducing the number of kernel crossings.

Stack Management

There are a number of related approaches to the problem of preallocating large stacks. Some functional languages, such as Standard ML of New Jersey [3], do not use a call stack at all; rather, they allocate all activation records on the heap. This approach is reasonable in the context of a language that uses a garbage collector and that supports higher-order functions and first-class continuations [4]. However, these features are not provided by the C programming language, which means that many of the arguments in favor of heap-allocated activation records do not apply in our case. Furthermore, we do not wish to incur the overhead associated with adding a garbage collector to our system; previous work has shown that Java's general-purpose garbage collector is inappropriate for high-performance systems [33].

A number of other systems have used lists of small stack chunks in place of contiguous stacks. Bobrow and Wegbreit describe a technique that uses a single stack for multiple environments, effectively dividing the stack into substacks [8]; however, they do not analyze the program to attempt to reduce the amount of run-time checks required. Olden, which is a language and runtime system for parallelizing programs, used a simplified version of Bobrow and Wegbreit's technique called "spaghetti stacks" [9]. In this technique, activation records for different threads are interleaved on a single stack; however, dead activation records in the middle of the stack cannot be reclaimed if live activation records still exist further down the stack, which can allow the amount of wasted stack space to grow without bound. More recently, the Lazy Threads project introduced stacklets, which are linked stack chunks for use in compiling parallel languages [18]. This mechanism provides run-time stack overflow checks, and it uses a compiler analysis to eliminate checks when stack usage can be bounded; however, this analysis that does not handle recursion as Capriccio does, and it does not provide tuning parameters. Cheng and Blelloch also used fixed-size stacklets to provide bounds on processing time in a parallel, real-time garbage collector [11].

Draves et al. [14] show how to reduce stack waste for kernel threads by using continuations. In this case, they have eliminated stacks entirely by allowing kernel threads to package their state in a continuation. In some sense, this approach is similar to the event-driven model, where programmers use "stack ripping" [1] to package live state before unwinding the stack. In the Internet servers we are considering, though, this approach is impractical, because the relatively large amount of state that must be saved and restored makes this process tedious and error-prone.

Resource-Aware Scheduling

Others have previously suggested techniques that are similar to our resource-aware scheduler. Douceur and Bolosky [13] describe a system that monitors the progress of running applications (as indicated by the application through a special API) and suspends low-priority processes when it detects thrashing. Their technique is deliberately unaware of specific resources and hence cannot be used with as much selectivity as ours.

Fowler et al. [16] propose a technique that is closer to ours, in that they directly examine low-level statistics provided by the operating system or through hardware performance counters. They show how this approach can be used at the application level to achieve adaptive admission control, and they suggest that the kernel scheduler might use this information as well. Their technique views applications as monolithic, however, so it is unclear how the kernel scheduler could do anything other than suspend resource intensive processes, as in [13]. Our blocking graph provides the additional information we believe the scheduler needs in order to make truly intelligent decisions about resources.

7. FUTURE WORK

We are in the process of extending Capriccio to work with multi-CPU machines. The fundamental challenge provided by multiple CPUs is that we can no longer rely on the cooperative threading model to provide atomicity. However, we believe that information produced by the compiler can assist the scheduler in making decisions that guarantee atomicity of certain blocks of code at the application level.

There are a number of aspects of Capriccio's implementation we would like to explore. We believe we could dramatically reduce kernel crossings under heavy network load with a batching interface for asynchronous network I/O. We also expect there are many ways to improve our resource-aware scheduler, such as tracking the variance in the resource usage of blocking graph nodes and improving our detection of thrashing.

There are several ways in which our stack analysis can be improved. As mentioned earlier, we use a conservative approximation of the call graph in the presence of function pointers or other language features that require indirect calls (e.g., higher-order functions, virtual method dispatch, and exceptions). Improvements to this approximation could substantially improve our results. In particular, we plan to adapt the dataflow analysis of CCured [27] in order to disambiguate many of the function pointer call sites. When compiling other languages, we could start with similarly conservative call graphs and then employ existing control flow analyses (e.g., the 0CFA analyses [34] for functional and object-oriented languages languages, or virtual function resolution analyses [30] for object-oriented languages).

In addition, we plan to produce profiling tools that can assist the programmer and the compiler in tuning Capriccio's stack parameters to the application's needs. In particular, we can record information about internal and external wasted space, and we can gather statistics about which function calls cause new stack chunks to be linked. By observing this information for a range of parameter values, we can automate parameter tuning. We can also suggest potential optimizations to the programmer by indicating which functions are most often responsible for increasing stack size and stack waste.

In general, we believe that compiler technology will play an important role in the evolution of the techniques described in this paper. For example, we are in the process of devising a compiler analysis that is capable of generating a blocking graph at compile time; these results will improve the efficiency of the runtime system (since no backtraces are required to generate the graph), and they will allow us to get atomicity for free by guaranteeing statically that certain critical sections do not contain blocking points. In addition, we plan to investigate strategies for inserting blocking points into the code at compile time in order to enforce fairness.

Compile-time analysis can also reduce the occurrence of bugs by warning the programmer about data races. Although static detection of race conditions is challenging, there has been recent progress due to compiler improvements and tractable whole-program analyses. In nesC [17], a language for networked sensors, there is support for atomic sections, and the compiler understands the concurrency model. It uses a mixture of I/O completions and run-to-completion threads, and the compiler uses a variation of a call graph that is similar to our blocking graph. The compiler ensures that atomic sections reside within one edge on that graph; in particular, calls within an atomic section cannot yield or block (even indirectly). This kind of support would be extremely powerful for authoring servers. Finally, we expect that atomic sections will also enable better scheduling and even deadlock detection.

8. CONCLUSIONS

The Capriccio thread package provides empirical evidence that fixing thread packages is a viable solution to the problem of building scalable, high-concurrency Internet servers. Our experience with writing such programs suggests that the threaded programming model is a more useful abstraction than the event-based model for writing, maintaining, and debugging these servers. By decoupling the thread implementation from the operating system itself, we can take advantage of new I/O mechanisms and compiler support. As a result, we can use techniques such as linked stacks and resource-aware scheduling, which allow us to achieve significant scalability and performance improvements when compared to existing thread-based or event-based systems.

As this technology matures, we expect even more of these techniques to be integrated with compiler technology. By writing programs in threaded style, programmers provide the compiler with more information about the high-level structure of the tasks that the server must perform. Using this information, we hope the compiler can expose even more opportunities for both static and dynamic performance tuning.

9. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 Usenix ATC*, June 2002.

[2] 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):53–79, February 1992.

[3] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, 1991.

[4] A. W. Appel and Z. Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, Jan 1996.

[5] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN - an extensible microkernel for application-specific operating system services. In *ACM SIGOPS European Workshop*, pages 68–71, 1994.

[6] M. W. Blasgen, J. Gray, M. F. Mitoma, and T. G. Price. The convoy phenomenon. *Operating Systems Review*, 13(2):20–25, 1979.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[8] D. G. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Communications of the ACM*, 16(10):591–603, Oct 1973.

[9] M. C. Carlisle, A. Rogers, J. Reppy, and L. Hendren. Early experiences with Olden. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing (LNCS)*, 1993.

[10] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[11] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, 2001.

[12] J. Cordina. Fast multithreading on shared memory multiprocessors. Technical report, University of Malta, June 2000.

[13] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Symposium on Operating Systems Principles*, pages 247–260, 1999.

[14] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the13th ACM Symposium on Operating Systems Principle*, pages 122–136. Association for Computing Machinery SIGOPS, 1991.

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for

application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[16] R. Fowler, A. Cox, S. Elnikety, and W. Zwaenepoel. Using performance reflection in systems software. In *Proceedings of the 2003 HotOS Workshop*, May 2003.

[17] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[18] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads, Stacklets, and Synchronizers: Enabling primitives for compiling parallel languages. In *Third Workshop on Langauges, Compilers, and Run-Time Systems for Scalable Computers*, 1995.

[19] T. Hun. Minimal Context Thread 0.7 manual. http://www.aranetwork.com/docs/mct-manual.pdf, 2002.

[20] B. LaHaise. Linux AIO home page. http://www.kvack.org/ blah/aio/.

[21] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Second Inernational Symposium on Operating Systems, IR1A*, October 1978.

[22] J. Lemon. Kqueue: A generic and scalable event notification facility. In *USENIX Technical conference*, 2001.

[23] D. Libenzi. Linux epoll patch. http://www.xmailserver.org/ linux-patches/nio-improve.html.

[24] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. Technical Report TR-90-09-05, University of Washington, 1990.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.

[26] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, 2304:213–229, 2002.

[27] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *The 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128–139. ACM, Jan. 2002.

[28] J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[29] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[30] H. D. Pande and B. G. Ryder. Data-flow-based virtual function resolution. *Lecture Notes in Computer Science*, 1145:238–254, 1996.

[31] W. Pang and S. D. Goodwin. An algorithm for solving constraint-satisfaction problems.

[32] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[33] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein. Java support for data-intensive systems: Experiences building the Telegraph dataflow system. *SIGMOD Record*, 30(4):103–114, 2001.

[34] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.

[35] E. Toernig. Coroutine library source. http://www.goron.de/˜froese/coro/.

[36] Unknown. Accellerating Apache project. http://aap.sourceforge.net/.

[37] Unknown. State threads for Internet applications. http://state-threads.sourceforge.net/docs/st.html.

[38] J. R. von Behren, E. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, , and D. Culler. Ninja: A framework for network services. In *Proceedings of the 2002 Usenix Annual Technical Conference*, June 2002.

[39] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 2003 HotOS Workshop*, May 2003.

[40] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, Decemeber 1995.

[41] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.