The Performance of an Object-Oriented Threads Package

John E. Faust and Henry M. Levy
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
USA

Abstract

Presto is an object-oriented threads package for writing parallel programs on a shared-memory multiprocessor. The system adds thread objects and synchronization objects to C++ to allow programmers to create and control parallelism. Presto's object-oriented structure, along with its user-level thread implementation, simplifies customization of thread management primitives to meet application-specific needs.

The performance of thread primitives is crucial for parallel programs with fine-grained structure; therefore, the principal objective of this effort was to substantially improve Presto's performance under heavy loads without sacrificing the benefits of its object-oriented interface. We discuss design and implementation issues for shared-memory multiprocessors, and the performance impact of various designs is shown through measurements on a 20-processor Sequent Symmetry multiprocessor.

1 Introduction

Explicit support for threads (multiple streams of execution within a single address space) has become common in experimental and commercial operating systems for both uniprocessor and multiprocessor architectures. For example, Mach [Accetta et al. 86] and Topaz [Thacker et al. 88] provide explicit kernel-level thread support. On a uniprocessor, an application utilizes threads simply as a program structuring tool, or possibly to overlap I/O or other asynchronous events with processing. On a multiprocessor, however, the various threads of an application can execute simultaneously on different processors, providing true parallel execution of the application. A substantial speedup can thus be realized in properly written applications.

While threads can be supported by the kernel, they can also be provided completely at the user level. In either case, the thread interface is typically presented as a collection of callable library routines, and thus the implementation is transparent to the programmer. However, the difference between kernel-implemented and user-implemented threads may be crucial to parallel programs with fine- or medium-grained structure. User-level threads can provide higher performance because kernel calls are avoided; the lighter weight the thread mechanism, the more freely the programmer can use threads to achieve speedup in parallel programs. Furthermore, user-level threads permit application-specific customization.

At the University of Washington we have developed two different user-level threads packages: Presto and FastThreads. Presto [Bershad et al. 88a, Bershad et al. 88b] is an object-oriented environment that allows the programmer to customize the threads package to reflect the specific needs of the application. Presto is implemented in C++ [Stroustrup 86]. FastThreads, which is implemented in C, evolved from the work of Anderson [Anderson et al. 89, Anderson 90]; it has a less flexible interface than Presto but provides higher performance. On the Sequent Symmetry, Presto is nearly two orders of magnitude faster than the process-based parallelism of Sequent's Dynix operating system, and FastThreads

This work was supported in part by the National Science Foundation under Grants No. CCR-8700106, CCR-8907666, and DCR-8619663, and by Digital Equipment Corporation's Graduate Engineering Education Program.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 089791-411-2/90/0010-0278...$1.50

is another order of magnitude faster than Presto.

Anderson achieved performance improvement over Presto through several means, including fine tuning through the use of C, the reduction of layering in the implementation, and the reduction of synchronization overhead through the use of more complex data structures. Our experience with Presto, however, has shown a number of advantages to its object-oriented interface. Therefore, the objective of the current work was to improve the performance of Presto by integrating some of the more successful ideas in FastThreads while maintaining Presto's object-oriented interface. The ideal goal was to achieve the same level of performance as FastThreads, or, failing that, to understand the intrinsic costs and limitations imposed by Presto's object-oriented approach and its use of C++.

This paper describes Presto, the techniques used to improve Presto's performance, and results of applying those techniques. It demonstrates the levels of performance that can be achieved using an object-oriented threads package. The following section presents a brief overview of Presto and its object-oriented interface. Section 3 covers specific areas in which Presto performance was tuned: spinlocking, atomic integers, thread creation and deletion, and thread startup. In each part of section 3, the performance of the original version of Presto will be presented, problem areas will be identified, modifications will be discussed, and resultant performance increases will be displayed. Finally, section 4 presents conclusions and areas for further work.

All measurements reported in this paper were obtained on a Sequent Symmetry multiprocessor with 20 Intel 80386 processors. The Symmetry has shared memory and a shared bus and utilizes a write-back invalidation-based cache coherency scheme [Lovett & Thakkar 88].

2 An Overview of Presto

Presto is a system designed to simplify parallel programming on a shared-memory multiprocessor through the use of object-oriented abstractions. The system applies our experience with distributed object-oriented systems [Almes et al. 85, Jul et al. 88] to multiprocessors. The basic idea behind Presto is the encapsulation of parallelism. That is, in conventional object-oriented systems, an object hides both its representation and its implementation. In Presto, an object hides its execution as well; the implementor of an object may choose either a parallel or sequential execution for an operation, but the choice is invisible to the invoker of that object.

The Presto programming environment consists of C++ with the addition of several classes useful for writing parallel programs. In particular, these classes provide the programmer with comfortable abstractions for dealing with parallelism and concurrency. Programmers write in C++, creating C++ objects of their own, and using Presto-provided classes (or customized versions of those classes) for parallelism.

Thread objects (threads) are the building blocks of Presto parallel programs. As the basic unit of execution, threads conceptually consist of a program counter and a stack of invocation records. There are two essential operations that can be performed on a thread. A thread can be *created*, allowing the creator to specify the thread's qualities, such as its name and storage requirements. Once created, a thread object can be started by invoking its *start* operation. The *start* invocation specifies as parameters an object, an operation in that object, and zero or more parameters to that operation; the started thread object then executes the specified operation *in parallel* with the invoking thread. Invocations in Presto are always synchronous, but a new thread can always be created to perform the equivalent of an asynchronous invocation.

When a Presto program is loaded, it receives one thread for its main program execution. The main program then creates more thread objects for parallelism, and as other objects are invoked, they may create more thread objects to do their work, and so on. All Presto objects execute within a single address space shared by all processors actively executing the application. Threads synchronize through the use of Presto-provided synchronization classes, such as relinquishing locks, non-relinquishing locks, and monitors [Hoare 74].

In Presto, threads are scheduled and controlled by special scheduler and processor objects. The system maintains a single scheduler object to keep track of all runnable threads. The scheduler object allows ready threads to be inserted and removed from a pool of ready threads according to the current scheduling discipline (the scheduling discipline can be easily changed using normal C++ inheritance mechanisms). Each allocated processor is represented by its own processor object. The processor object creates and executes a "scheduler" thread whose only task is to request a runnable thread from the scheduler object. When a runnable thread is obtained, the processor object stops running the scheduler thread and starts running the new thread. When the newly run thread blocks or terminates, the scheduler thread resumes and continues to check for more runnable threads. When a thread blocks, for ex-

ample on a synchronization object, its next execution may occur on a different processor.

On systems like Dynix, Presto runs a thread by mapping it onto a preexisting Dynix process created during Presto initialization. The Dynix kernel then maps the process onto a physical processor for execution. In Dynix, processes can be permanently bound to a processor, so kernel-level process scheduling need only be performed once.

Because Presto is implemented entirely at the user level, any of its objects, including thread objects and scheduler objects, are open to customization by the programmer. Thus, thread objects can be easily customized for performance instrumentation, or scheduler objects can be customized to provide specialized scheduling algorithms. For example, in a parallel simulation system built from Presto [Wagner et al. 89], the standard scheduler was customized to handle deadlock detection during idle periods. Performance instrumentation was added to Presto's thread and locking constructs to integrate support for Quartz [Anderson & Lazowska 90], a parallel program tuning tool, into Presto. Customization is accomplished through the normal C++ inheritance mechanisms – the programmer creates a more specialized subclass of an existing Presto class, redefining the implementation of operations defined on that class, or adding new operations as required.

It is important to contrast Presto to recent efforts in object-oriented concurrent programming (e.g., [Agha & Hewitt 87, America 87, Yokote & Tokoro 87, Kaiser et al. 89, Agha et al. 88]). In general, these efforts have been focused on programming support for concurrency *within* objects. Presto, on the other hand, has focused on the implementation of inexpensive parallelism. Our objective was to build an object-oriented framework to simplify parallel programming of "conventional" applications, in order to maximize speedup on medium-scale multiprocessors (those with 5 to 30 processors).

The crucial issue in such an environment is performance, and in particular, the cost of parallelism. For example, in Sequent's Dynix operating system, the entities for parallel programming are Dynix processes, which cost on the order of 5 to 10 milliseconds to create. This leads to a static programming style with the number of processes equal to the number of processors. If parallelism can be provided cheaply, however, programmers can use a level of parallelism that is natural to the application, as opposed to that dictated by hardware and software constraints.

3 Improving the Performance of Presto

While the first implementation of Presto achieved close to two orders of magnitude improvement over Dynix processes, there was still much room for improvement. Of interest were both the latency of operations (on one processor) and the speedup achieved as more processors are added. In order to achieve another order of magnitude speedup, Anderson built a more streamlined thread system, called FastThreads, in C [Anderson et al. 89, Anderson 90]. While FastThreads has lower latencies and better performance under loads, it lacks several advantages of Presto, namely the object-oriented interface and the simple methods for application programmers to customize and add to the base structure.

As previously stated, the objective of this work was to improve Presto's performance without sacrificing its object-orientation. This section describes some of the changes made to the original Presto implementation in order to improve its performance characteristics, and presents the performance achieved by those means. In particular, we examine the implementation of primitives for locking (spinlocking, atomic integers) and thread management.

3.1 Spinlocking

Spinlocking, or busy-waiting on a semaphore, is a fundamental technique for ensuring mutual exclusion on a shared-memory multiprocessor. On most concurrent programming systems, all locking operations cause a thread to be queued when a lock request finds the lock busy, thus permitting another thread to run. Note, however, that the enqueueing of a thread requires a queue manipulation operation that must be protected by a spinlock; therefore, spinlocks are always required at some level, either by the thread system or for use by the application programmer.

Spinlocking is often criticized as a potential waste of CPU cycles. Certain applications, however, can achieve noticeable performance improvements when busy-waiting for a lock rather than relinquishing the processor. For example, if critical sections are small, then the average time spent busy-waiting for the semaphore may be smaller than the time needed to enqueue the thread in a wait queue. Also, if a thread relinquishes its processor to wait for a lock, it may resume execution on a different processor. Numerous cache misses will then be incurred on the new processor, even

though the data may still be "hot" in the cache on the original processor. Spinlocking allows for more effective utilization of the cache, since the thread remains on the processor where its data and instructions have been cached.

The original version of Presto provides spinlocking functionality based on a test-and-set strategy. Processors spin on a single, shared location until they successfully acquire the lock. The problem with this approach is that the normal memory requests of the processor currently holding the lock are delayed dramatically by the cache coherency traffic caused by the test-and-set instructions generated by each spinning processor (each "set" operation results in invalidation and refetch bus traffic by all other processors). For short critical sections, the delay in servicing the lock holder's memory requests dominates overall performance [Anderson 90].

FastThreads spinlocks are both lower in latency and more tolerant of high contention than the spinlocks of the original version of Presto. In [Anderson 90], five different spinlocking alternatives were evaluated. The best overall performance was achieved by using a queue-based locking strategy. Instead of using a single semaphore on which all processors spin, each processor attempting to acquire a lock adds itself to the tail of a queue and then spins on a semaphore provided by its predecessor in the queue. When a processor releases the lock, it dequeues itself and then sets its private semaphore to notify the next waiting processor. Unique sequence numbers can be used to emulate queueing without actually executing queueing instructions.

This method of spinlocking performs well when there is high contention for the lock. Each processor spins on a separate location instead of on a single, shared location as in the test-and-set approach. If each processor's private location falls within a separate cache block, cache coherency bus traffic related to the lock acquisition and release is virtually eliminated. Lock release is reduced to setting a location – a single, non-atomic instruction. On the other hand, queue-based spinlocking results in slightly higher lock latency when there is low contention, since more instructions must be executed than in the simple test-and-set case.

We modified Presto by adding queue-based spinlocking as a separate C++ class (HP_Spinlock), in order to allow both regular test-and-set spinlocks and queue-based spinlocks to coexist. The version of queue-based spinlocking we selected was devised by [Graunke 88]. This version creates an implicit queue by passing information about the previous lock requestor to each new lock requestor. The information passed allows the new lock requestor to locate the semaphore of its predecessor in the queue.

A benchmark was prepared in order to facilitate comparison of spinlocking performance in Presto and FastThreads. The benchmark computes the elapsed time required for various numbers of processors (one thread on each processor) to cooperatively execute a critical section one million times. In the test, each thread acquires the spinlock, executes a short (2-3 instruction) critical section, releases the lock, and then computes. The compute phase is used to ensure fairness – without it, the processor that just acquired and released the lock is the processor that is most likely to successfully acquire the lock, because the lock is still in that processor's cache. A compute phase length of approximately 20 μsecs[1] was utilized for all test runs.

Figure 1 shows the results of the spinlocking benchmark for the three systems. The curve for the original version of Presto shows how performance degrades due to the delay in servicing the lock holder's memory requests caused by the test-and-set based cache coherency traffic. Note that the point at which performance begins to degrade would increase with the length of the computation phase following the critical section, since a longer computation period would make it increasingly likely that a given thread would be computing rather than contending for the lock. In comparison, the curves for both FastThreads and for Presto with the high-contention spinlock (curve *HP_Spinlock*) show a levelling off of performance but no degradation as contention increases. Both curves show slightly higher latency than the original Presto spinlock when there is little contention for the lock, due to the greater number of instructions executed.

3.2 Atomic Integers

Presto includes class AtomicInt to allow the programmer to easily utilize protected integer operations, e.g., incrementation of a shared integer counter. The implementation of class AtomicInt includes an integer value, an object of class Spinlock to protect the integer, and a collection of member functions, all of which lock the spinlock, perform an integer operation on the value, and then release the lock.

FastThreads includes no such built-in functionality. The programmer can construct the equivalent by asso-

[1] This benchmark is unrealistic in that few applications would actually be designed to request a lock 20 μsecs after releasing it, however, the objective here is to stress the spinlock implementation by providing an environment of high contention.

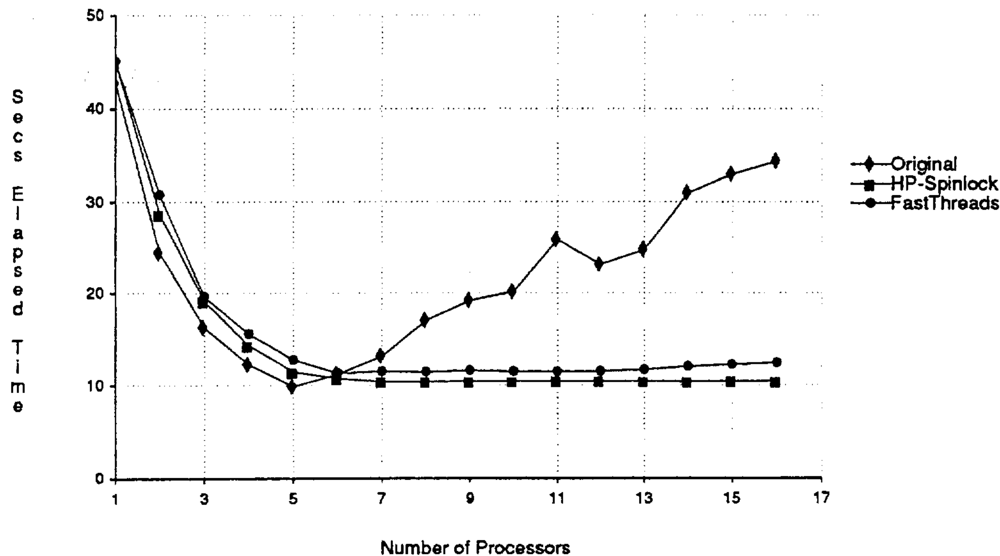

Figure 1: Elapsed time to acquire one million locks.

ciating a spinlock with an integer value and providing a collection of functions which modify the integer under spinlock protection.

We added class HP_AtomicInt to Presto, utilizing the high-contention spinlock described in the preceding section to protect the integer value. Our first implementation derived class HP_AtomicInt from the base AtomicInt class. The lock and unlock functions in the base class were made "virtual" to allow replacements for them to be defined in the derived class. The derived class replaced these functions with equivalents that were specific to the HP_Spinlock class.

The spinlocking benchmark from the previous section was modified slightly (replace lock acquisition, critical section, and lock release with atomic integer incrementation). The new benchmark measures the elapsed time for various numbers of processors (one thread runs on each processor) to increment the atomic integer one million times.

The curve labelled *Original* in figure 2 shows the results of the benchmark for the original AtomicInt class. The performance degradation observed in the original implementation is due to the impact of cache coherency traffic caused by the test-and-set instruction, as discussed in the previous section. The curve labelled *HP-Derived* shows that the high-contention spinlock has eliminated this bottleneck.

Note, however, that the *HP-Derived* curve in figure 2 differs significantly from the *HP-Spinlock* curve in figure 1. Since the benchmarks are nearly identical, the difference must be due to the HP_Derived implementation itself. Closer examination reveals that virtualizing the lock and unlock functions in the base AtomicInt class prevents our version of C++ (AT&T V1.2) from following the *inline* hint given for the replacement lock and unlock functions in the derived class. The observed performance difference is simply due to the two extra routine calls required in the HP_Derived benchmark.

We reimplemented class HP_AtomicInt as its own base class in order to allow the new lock and unlock functions to be successfully inlined. The curve labelled *HP-Base* in figure 2 shows the benchmark results for the new class. Note that this curve compares favorably with the HP_Spinlock results of figure 1.

3.3 Thread Creation and Deletion

Efficient thread implementation was one of the principal goals of Presto's first implementation. However, even with the care taken in this part of the design, thread creation performance suffered under heavy loads. In order to understand the limitations of thread creation performance, we must first describe some of the details of Presto's thread implementation.

A Presto thread object contains a moderate amount of information about the state and execution context of a thread of execution. Information stored here includes the thread ID, a pointer to the thread's stack,

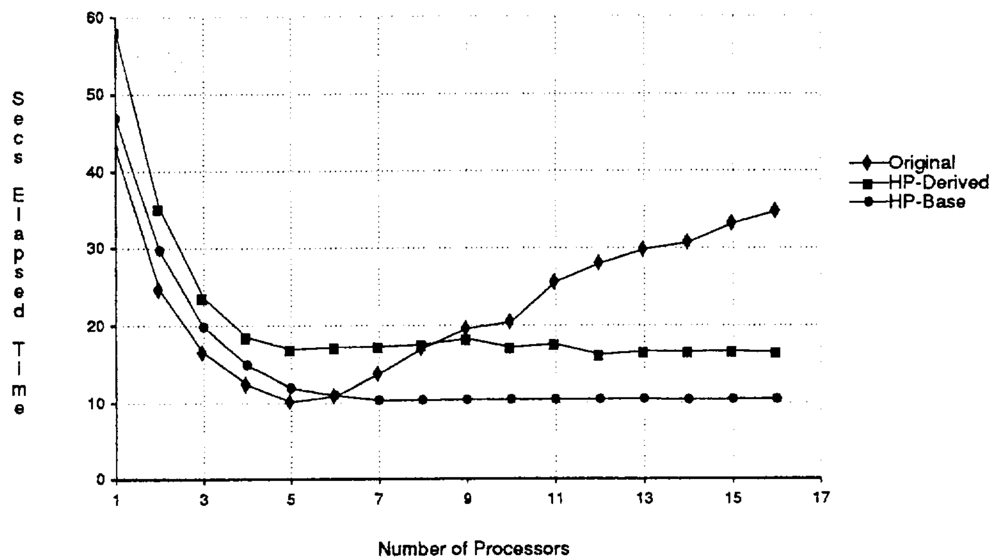

Figure 2: Elapsed time to increment atomic integer one million times.

the stack size, the current stack and frame pointers, the thread state, etc. Dynamic thread objects, like all dynamic objects in C++, are allocated from the heap. In order to avoid the overhead of heap allocation or deallocation each time a dynamic thread object is created or destroyed, Presto maintains a cache of thread objects. When a thread terminates, Presto places the thread object in this cache for potential reuse. When a new thread object is created, Presto first checks the cache, allocating from the heap only when the cache is empty. A threshold value prevents a thread-intensive application from consuming undue amounts of virtual memory. The thread cache is implemented as a shared queue, protected by a spinlock.

Each Presto thread object has a private stack, and thread stacks are also cached for potential reuse. When a stack is "deleted", it is moved into a cache of stack objects. Stack "creation" is a bit more complex: since stacks can vary in size a search must be performed to find a stack of the proper size. Allocation occurs only when a stack of proper size is not found in the cache. As in the thread cache, the size of the stack cache is limited by a threshold value, and the cache is implemented as a shared, protected queue.

In [Anderson et al. 89], locks protecting the shared caches of threads and stacks were found to be a bottleneck in thread creation. To eliminate this bottleneck, separate *per-processor* caches were used. Since the caches were private to a given processor, lock protection was no longer needed. A locked, central thread cache was also maintained to help keep the per-processor caches balanced.

We modified Presto to include separate per-processor caches of reusable thread templates. When a thread object is created in this new version of Presto, a thread template is pulled from the creating processor's local thread cache if possible. If no thread templates are available in the local cache, an attempt is made to move a group of threads from a central, shared cache to the local cache. If a thread template is still not available, then a new thread template will be allocated from the heap. When a thread object terminates, its thread template is placed in the executing processor's local cache. If the resultant cache size exceeds a threshold value, a group of thread templates are moved to the centralized cache. In order to improve the performance of the initial thread creation requests, threads are preallocated and placed in the local thread caches during Presto initialization.

Per-processor caches of reusable stacks were unfortunately more complex to implement. The flexibility of variably-sized stacks makes balancing of the stack cache more difficult and expensive to manage. Searching the stack cache for a stack of the proper size adds to the latency of thread creation. To avoid these problems, stacks were forced to be of fixed size, eliminating the search for a stack of the proper size (and allocation if no stack is found). They were also bound to a thread once, when the thread was originally allocated, eliminating the need to maintain stack freelists and thus the

need to balance those lists.

These changes decrease the latency of thread creation, but they are not without a cost. Forcing all stacks to be of one size increases memory costs, since threads requiring less stack space are unable to specify that a smaller stack be used. Binding a stack to a thread when the thread is allocated may exhaust memory if many threads are allocated before being used – in original Presto, stacks were bound only when a thread was about to be executed. Our modifications trade off flexibility and reduced memory cost for decreased latency in thread creation. We believe that thread creation is now nearly fast enough to eliminate the need for the programmer to preallocate threads that will be used later.

A benchmark was created to measure the performance of thread creation and deletion. The benchmark measures the elapsed time required for various numbers of processors (one thread per processor) to cooperatively create and delete one million threads. None of the created threads are started, so this benchmark excludes all activities related to thread scheduling and startup.

Figure 3 shows the benchmark results for the various implementations. The curve for the original version of Presto shows how performance degrades due to contention for the spinlocks protecting the shared caches of threads and stacks. The local thread cache implementation (curve *LTQ*) eliminates this lock contention and demonstrates good performance for up to 13 processors. The local thread caches with fixed stacks (curve *LTQ-FSS*) version shows no performance degradation. An inverse plot of this curve, giving throughput instead of elapsed time, would show an almost linear increase in the rate of thread creation.

3.4 Thread Startup

In Presto, thread startup causes the specified thread object to begin executing an operation in parallel with the invoking thread. In order to begin execution, the started thread must be added to the Presto scheduler object's pool of ready threads and eventually dispatched to an available Presto processor object. In original Presto, the pool of ready threads is implemented as a shared queue protected by a spinlock.

Anderson found the use of a shared ready queue to be a significant bottleneck. To avoid this bottleneck, FastThreads utilizes separate *per-processor* queues of ready threads. In this case, however, the per-processor queues require spinlock protection, since if a processor finds no ready threads in its local queue, it looks in the queues of its neighbors.

The benchmark from the preceding section was modified slightly to start the created threads. The new benchmark measures the elapsed time required for various numbers of processors to create, start, and delete one million "null" thread objects. Figure 4 shows the benchmark results for three versions of Presto. The curve for the original version of Presto shows that performance begins to level out at about 8 processors, degrading slightly from about 11 processors onwards. The curve labelled *LTQ-FSS* shows how the thread creation enhancements from the preceding section have improved performance. However, this version also shows a levelling-off and eventual degradation in performance. In both curves, performance is limited by the remaining bottleneck: the use of a single, shared thread ready queue.

We constructed a modified version of Presto which utilized per-processor ready queues as in FastThreads. A regular test-and-set based spinlock was used to protect each local queue (it was assumed that high contention for any one local queue would be rare, and the test-and-set spinlock has lower latency). The curve labelled *LTQ-FSS-LRQ* in figure 4 shows how per-processor ready queues eliminate the performance degradation caused by contention for the lock protecting the single, shared ready queue. The figure shows improved performance from 2 processors on upwards, indicating that in our benchmark, contention for the single, shared ready queue was a factor with as few as two processors!

Figure 5 contrasts the benchmark performance of this newest version of Presto (*LTQ-FSS-LRQ*) with that of original Presto and with FastThreads. The figure shows that although a large improvement of performance over original Presto has been made, there is still quite a bit of work to be done in order to achieve the performance of FastThreads. While the *LTQ-FSS-LRQ* version of Presto has improved throughput to within a factor of three of FastThreads, latency is still a factor of seven worse than that of FastThreads. Note that in figure 3 we measured the latency of thread creation and deletion in modified Presto at approximately 70 μsecs. In figure 4 we find that addition of the thread startup operation increases latency to 270 μsecs! Thread startup is obviously a primary contributor to the seven-time latency difference observed above.

We suspected that the high thread startup latency was due to the overhead of the extra layers of abstraction built in to the class hierarchy for Presto's thread startup operation. Execution profiling of the startup code path showed that our suspicions were correct. For

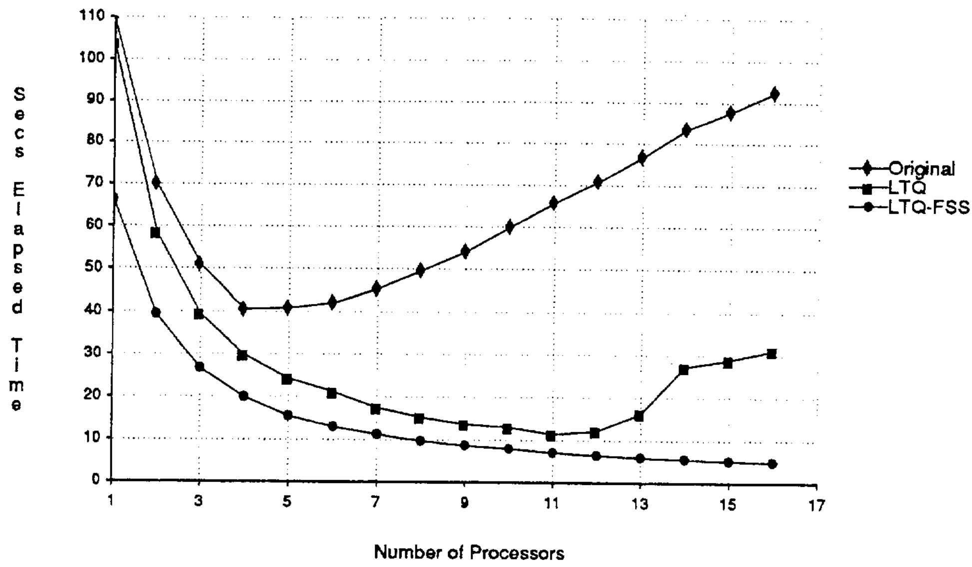

Figure 3: Elapsed time to create and delete one million threads.

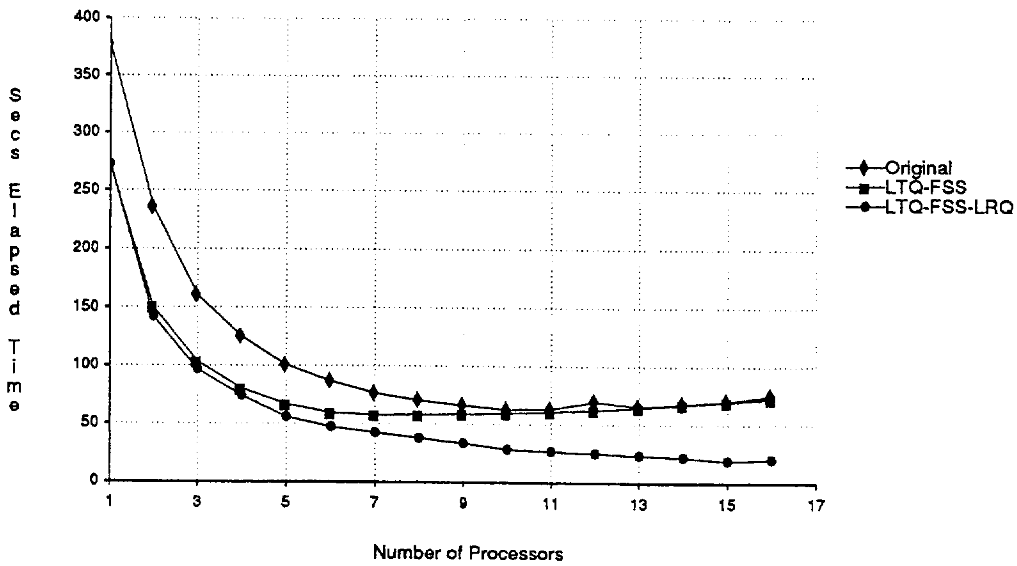

Figure 4: Elapsed time to create, start, and delete one million null threads.

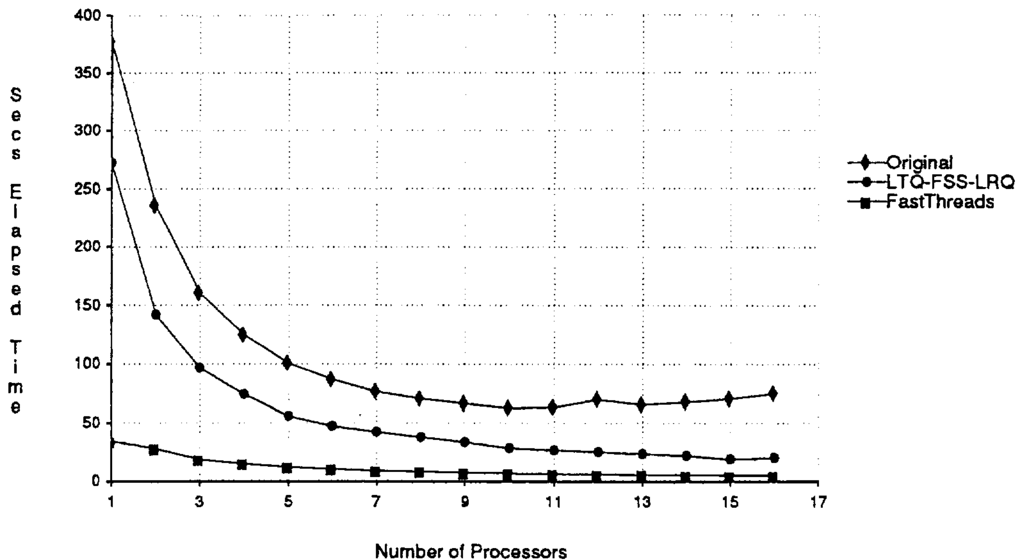

Figure 5: Elapsed time to create, start, and delete one million null threads.

example, a substantial portion of thread startup time (14%) is spent within the member functions of an object used to temporarily store the arguments that are passed to the routine a thread will execute. This object stores the thread arguments until a thread stack has been allocated. An additional 24% of the time spent in starting a thread is due to the cost of adding and removing a thread from the pool of ready threads maintained by the Presto scheduler object. Extra layers of abstraction are used here in order to facilitate the replacement of the default scheduling discipline (simple FIFO queueing) with whatever discipline the application requires.

Anderson achieved high-performance thread startup by limiting generality, allowing for a minimization of thread state and thus a reduction in thread startup latency. The latency of Presto thread startup could be decreased in the same way. For example, the thread argument storage object could be eliminated, and some of the layers of abstraction in the Presto scheduler object could be removed. However, one of our goals in this work was to maintain compatibility with Presto's original object-oriented interface, and these changes would not be consistent with that goal.

Constrained from further improving Presto's performance by our goal of interface compatibility, we decided to see if we could apply Anderson's techniques to implement our own low latency thread startup operation. Using FastThreads as a model, we implemented a new object-oriented threads package in C++. The new package is less general and thus less flexible than Presto, but as in FastThreads, by limiting generality it reduces thread state and thus thread management code path length. The resultant package offers much better thread startup performance – using the new package, we measured the latency of thread creation, startup and deletion at 76.74 μsecs, a factor of five improvement over original Presto, and within a factor of 1.9 of the performance of FastThreads.

Presto's flexible, object-oriented structure results in longer code paths caused by an increase in the number of layers of abstraction. For operations such as thread startup, the result is higher latency. FastThreads demonstrated that limiting generality can improve performance by reducing thread state and thus code path length. Our work has shown that by carefully choosing where to limit generality, it is possible to construct an object-oriented threads package with low latency operations.

4 Conclusions

This paper has shown that it *is* possible to provide a threads package with both an extensible, object-oriented interface and high performance primitives. Integration of a spinlocking strategy specifically designed to reduce cache coherency bus traffic allows high-contention spinlocking and atomic integer primitives to be provided. Several modifications led to more efficient thread creation and deletion: utilization of un-

locked, per-processor queues of thread templates, elimination of all overhead associated with binding stacks to threads, removal of unnecessary atomic integer operations, and general code cleanup. Use of per-processor queues of ready threads removed the bottleneck caused by a single shared ready queue, improving the performance of thread startup.

These changes resulted in a noticeable performance improvement in Presto. The new version of Presto provides the same spinlocking performance (in terms of both latency and throughput) as FastThreads, and Presto's atomic integer performance has been enhanced as well. The new version of Presto drops thread creation and deletion latency by 40% and increases throughput by a factor of 19 over original Presto. Thread startup performance has also been improved – we have decreased latency by 28% and increased throughput by a factor of four over original Presto.

Thread creation and startup performance in the new version of Presto still lags behind that of FastThreads. Throughput is within a factor of three, but latency is still worse by a factor of seven. We implemented a new object-oriented threads package in C++ that is much simpler than Presto, and obtained thread startup latency within a factor of 1.9 of that of FastThreads. This experiment showed that the bulk of Presto's thread startup latency is due to its flexible, customizable user interface, not to its object-orientation or its implementation in C++.

During our work the advantages of the object-oriented approach quickly became evident. Since objects are accessible only through a predefined interface, other objects are unable to make assumptions about an object's implementation. The implementation of an object can thus be changed with no impact to objects which depend on that object. We made substantial changes to the underlying implementation of multiple low-level Presto objects. The only impact of these changes on other objects was improved performance.

This work was constrained somewhat by our use of AT&T C++. Since this compiler simply preprocesses code into C, it is not well integrated with the underlying optimizing C compiler. This impacts the quality of code generated and thus impacts performance. It also restricts the ability to inline functions containing loops or assembler code. Conversion to a different implementation of C++ might address some of these problems.

A key area for further work is to continue to reduce the latency of Presto thread creation and startup. Further performance improvements in this area will require modification of Presto's current user interface in order to remove extra layers of abstraction. A different approach might be to complete the skeletal threads package used to prove that Presto's thread startup latency can be greatly reduced. This package could be augmented to provide more of the functionality of Presto, while still retaining the low-latency thread startup operation. The result of this work would be a robust, object-oriented threads package capable of supporting fine-grain parallelism.

Such a package would be quite useful to programmers of parallel applications. The object-oriented approach would allow programmers to derive their own specialized subclasses from the base classes provided by the threads package. The resultant application would be insulated from future changes to the implementation of any class used by the application. Support for fine-grain parallelism would allow the programmer to use parallelism dynamically as dictated by the application. The package could also be incorporated into compilers for languages with built-in support for parallelism. Object-oriented packages would be particularly useful in this environment, since they would would allow thread scheduling and preemption to be tailored to support specific language requirements. Support for fine-grain parallelism would allow such compilers to schedule smaller units of work for parallel execution.

The new version of Presto described in this paper is freely available through anonymous FTP from University of Washington on cs.washington.edu.

5 Acknowledgements

We would like to thank Tom Anderson, Brian Bershad, and Ed Lazowska for technical help on understanding Presto and FastThreads and for helpful reviews of this paper. We would also like to thank Reid Brown for his work on the software used to generate the graphs included in this paper.

References

[Accetta et al. 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and M. Y. A. Tevanian. MACH: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, June 1986.

[Agha & Hewitt 87] G. Agha and C. Hewitt. Concurrent object-oriented programming in Act-1. In A. Yonezawa and M. Tokoro, edi-

tors, *Object-Oriented Concurrent Programming*, pages 9–36. The MIT Press, 1987.

[Agha et al. 88] G. Agha, P. Wegner, and A. Yonezawa, editors. *Proceedings of the ACM Sigplan Workshop on Object-Based Concurrent Programming.* ACM Press, September 1988.

[Almes et al. 85] G. Almes, A. Black, E. Lazowska, and J. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, January 1985.

[America 87] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, 1987.

[Anderson & Lazowska 90] T. Anderson and E. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proceedings of the ACM SIGMETRICS Conference on Performance Measurement and Modeling of Computer Systems*, May 1990.

[Anderson 90] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[Anderson et al. 89] T. Anderson, E. Lazowska, and H. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[Bershad et al. 88a] B. Bershad, E. Lazowska, and H. Levy. Presto: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[Bershad et al. 88b] B. Bershad, E. Lazowska, H. Levy, and D. Wagner. An open environment for building parallel programming systems. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming Environments, Applications, and Languages*, July 1988.

[Graunke 88] G. Graunke. personal communication, 1988.

[Hoare 74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Kaiser et al. 89] G. Kaiser, S. Popovich, W. Hseush, and S. Wu. MELDing multiple granularities of parallelism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 147–166, July 1989.

[Lovett & Thakkar 88] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[Stroustrup 86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Thacker et al. 88] C. Thacker, L. Stewart, and E. Satterthwaite Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Wagner et al. 89] D. Wagner, E. Lazowska, and B. Bershad. Techniques for efficient shared-memory parallel simulation. In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, March 1989.

[Yokote & Tokoro 87] Y. Yokote and M. Tokoro. Concurrent programming in Concurrent-Smalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. The MIT Press, 1987.