The Design and Implementation of Efficient Thread-Based Parallelism

Dawson R. Engler*
M.I.T. Laboratory of Computer Science

Febuary 8 1994

Abstract

Efficient thread management allows operating systems and applications to effectively exploit parallelism. To aid designers in implementing an efficient thread package we record the evolution of a traditional stack-based thread package to a highly optimized "stateless" threads package. The effects of each optimization are discussed and quantified. Compared to the initial package, thread creation is made an order of magnitude more efficient, and thread execution 12-40 times faster, depending on the type of application. The final result is a new thread package, called Filaments. The cost of running a filament taking three arguments is approximately 6 instructions; creation is approximately 12. In addition, by exploiting the semantics of stateless threads, software based multithreading can be implemented with an efficiency competitive to a hardware implementation. Finally, Filaments is very portable; unlike other thread packages, it is completely written in a high-level language.

1 Introduction

Structuring modern operating systems and applications with threads allows them to overlay computations and thus extract high performance from parallelism. Obviously if thread primitives have a high overhead, achieving this goal will be difficult. Since generality is expensive, general purpose thread packages have high overhead, limiting their applicability. For example, thread creation in Pthreads [26] (an implementation of the general purpose POSIX thread specification), is over three orders of magnitude more costly than in the thread package described in this paper, Filaments. To reduce the expense of thread primitives, implementors concerned with efficiency often design packages optimized for their specific domain. To assist this process we present a comprehensive series of optimizations applied to an existing thread package. The effects of these optimizations are quantified using three different styles of thread usage: run-to-completion threads, iterative threads and fork/join threads.

*Most of this work was done when the author was at the University of Arizona. For correspondence: Dawson R. Engler, Massachusetts Institute of Technology, Laboratory of Computer Science, 545 Technology Square, Cambridge, MA 02139. Internet: engler@lcs.mit.edu

One result of our optimizations is the enabling of efficient fine-grain thread-based parallelism. Fine-grained parallelism is desirable for several reasons: architecture independence, ease of programming, ease of code generation and load-balancing potential. Unfortunately, while fine-grained parallelism has attractive attributes as a programming model and a target for code generation, conventional thread packages have been too inefficient. For example, Lin and Snyder [23] found a fine-grained implementation of Jacobi iteration using the Sequent Symmetry's parallel programming library to be 8 to 23 times slower than a coarse-grain one. More recent work, such as Chores [16], has been able to get efficient performance by automatically clustering fine-grained tasks into larger units, but executing fine-grained tasks independently is still 10 to 20 times slower, again for Jacobi iteration on a Sequent Symmetry. Using the approach discussed in this paper, fine-grained Jacobi iteration can be within 11% of a coarse-grained implementation.

The optimizations we discuss lie in two categories: *implementation effecting* and *semantic changing*. An example optimization of the first type is replacing the context-switch from an exiting thread to a as-yet-unrun thread by calling the new thread as a function, thereby eliminating the need to context-switch. An optimization of the second type is the complete elimination of stacks. The latter optimization greatly improves performance, but can only be applied when the programmer follows certain rules of threads usage. Trading ease of programming for performance is a decision that is often applied; for example, a weakly consistent memory [15], [7]. A compiler is sometimes used to implement the contract.

We applied the optimizations to a traditional stack-based thread package that was extracted from the SR language's runtime system [5]. The result is a novel thread package: *Filaments*. Filaments is a *stateless* thread package, only allowing blocking primitives that do not require a context-switch. Although Filaments is less general than traditional packages, in practice almost all parallel programs can be easily written using these more restrictive threads. The benefits of using Filaments are high performance and portability. The low overhead of Filaments' primitives allows programs to exploit fine-grained parallelism. The implementation is entirely written in C, which makes the package architecture-independent.

This paper presents two contributions: (1) a detailed exposition of the construction of an efficient thread package and (2) the explicit concept of stateless threads. Two important results of stateless threads are the ability to completely implement a thread package in a high level language and to have the cost of software based multithreading be comparable to that of a hardware implementation.

Many papers have discussed how to make threads efficient (e.g., [3], [2] and [30]) Our work synthesizes previous work on making threads efficient and extends it in several novel ways. Optimizations occur within a unified framework that allows their relative merits to be judged. Finally, the granularity of our discussion is concrete enough that implementors should have no trouble either in applying our ideas to new areas or in translating the presented optimizations into working code.

The rest of the paper is organized as follows. Section 2 describes the interface of the initial thread package, defines the three thread types we examine and enumerates the benchmarks used to measure the effects of our changes. Section 3 discusses the optimizations we applied to our stack-based thread package, as well as the fundamental limitations of a stack-based approach. Section 4 presents our most radical innovation—making threads stateless—and its implications for applications. Further optimizations are detailed and we also present the interface to Filaments. Section 5 discusses how the semantics of stateless threads can be exploited to allow very efficient multithreading. In Section 6 we compare related work. In Section 7 we draw our conclusions.

2 Experimental Details

In this section we describe the machine on which our tests were performed, the interface and implementation details of the initial stack-based thread package, and the three thread types we used to measure performance and discuss the benchmarks used to measure the effects of our changes.

The machine on which our tests were performed was a Silicon Graphics Iris 4D/340 multiprocessor, which has a 33 MHz clock, 64K byte instruction cache, 64K byte data cache, and 256K byte secondary data cache. We used the vendor supplied cc compiler on all tests, with the optimization flag "-O3".

The initial package was derived from SR's runtime system, and contains the core functions found in many operating systems, runtime systems, and thread libraries. Appendix A briefly describes its initial interface. For simplicity, common thread functions that were unnecessary for our benchmarks (e.g., conditional wait) have been eliminated. Reintroducing these primitives is not difficult.

The package does not support preemption except at barrier synchronization points. The reasoning is that a parallel program cannot complete until all work is done. If a thread needs to lock a shared data structure, it must spin until it acquires the lock. However, this cannot cause deadlock, because the lock must be held by another thread, which is executing and will not be preempted and hence will eventually release the lock.

The package was developed on a small-scale multiprocessor, and is small (approximately 300 lines of code) and simple. The thread package uses three queues. One is a central queue used to hold runnable threads. The remaining two are local queues that hold either blocked threads or available thread descriptors. The central queue enqueue and dequeue operations are locking. All local queue accesses are unlocked. A pool of heavyweight processes (called *servers*) is used to run threads. There is at most one server per processor. Each heavyweight server runs a simple scheduling algorithm that attempts to obtain work from the central queue and, upon success, context switches into the new thread. When a thread completes or blocks, it calls the scheduler function, and this cycle repeats. Thread creation consists of allocating both a control block and stack, and the construction of a context. The required memory is obtained from a central, locked free-list. If this free-list is depleted, memory is allocated using the system's shared-memory allocator.

The advantage of the above implementation is that it is very simple and allows effective load-balancing through the use of a central queue. The disadvantages are (1) the usual costs of a central structure: over-constraint of parallelism and contention and (2) a lack of any explicit mechanism for managing locality.

Name	Benchmark Description	Justification
null	Simple null creation and execution.	Tests the overhead of thread primitives.
Jacobi	Jacobi iteration, process per point.	Measures the overhead of synchronization primitives and the effect of repeated data manipulations on performance.
Fibonacci	Calculation of Fibonacci numbers using fork/join paradigm.	Measures overhead of the fork/join primitives.
matrix	Matrix multiply, process per point.	A large amount of data is touched, allowing the effects of each change to be seen from a caching perspective.

Figure 1: Description of benchmarks

Run-to-completion, fork/join and iterative threads are sufficient to implement a broad class of important problems. Therefore we have measured the utility of an optimizations based on it impact on these three thread types. The definitions, examples, and operations associated with each thread type are given below.

Run-to-completion (RTC) threads: RTC threads are threads that complete without blocking. They are used in problems such as matrix multiplication or matrix addition.

Fork/join threads are used in divide-and-conquer algorithms where a parent thread forks some number of children and then waits (blocks) until they complete. Examples are recursive Fibonacci calculations, quicksort and recursive FFT. Our initial implementation of fork/join is a traditional one. When a parent forks children, its counter of active children is atomically incremented. When a child completes its parent's counter is atomically decremented. A "join" blocks the parent until all of its children have completed. This is done by suspending the parent until all children have completed; the last child context-switches into the parent which then resumes execution.

Barrier threads are threads that execute and synchronize with their peers repeatedly. Barrier threads are used for such computations as Jacobi iteration and Gauss-Seidel relaxation. The package runs all threads and, as each thread blocks at the barrier, moves it to a local suspended queue. When all threads complete, the servers synchronize and a single server calls user provided sequential code to check for termination and to allow reinitialization of user data structures. If termination is detected, all of the suspended threads are moved to the free list and the servers then terminate. Otherwise, the suspended threads are moved back to the ready queue and another iteration is done. This barrier implementation is novel in that the cost of barrier synchronization is proportional to the number of servers, not the number of threads.

The four benchmarks used to measure the impact of the our optimizations are given in Figure 1. The parameters tested are the raw overhead of creation and execution (null), the effects of the changes on locality (matrix), barrier iteration overhead (Jacobi), and fork/join performance (Fibonacci).

Optimization	Description	Implication
Central Queue	Central ready queue	Simple load-balancing, but has contention and no locality.
Batch work	Remove work in chunks	Reduce Contention and primitive locality management. Needs two queue traversals.
Local Queue	Local queues	Locality, more efficient parallel create. Now need to do explicit locality management.
Disabled Load-Balance	Remove locks	Eliminate locking cost and bus contention.
Lazy Save	Special context-switching for exiting and new threads	When context-switching to new threads load only those registers that are necessary, at exit do not save any registers. Checks are put in the code to determine if thread is unrun.
Inexpensive Index	Inline queue operations and make queue structure size a power of two	Indexing can be done by shifts, and padding reduces contention.
Lazy Stack	Lazy stack allocation	Possibility of saving much memory. Adds overhead if large number of threads block and increases the size of thread descriptor.
Live Stack	Live stack reuse	Eliminate stack setup costs.

Figure 2: Optimizations applied to stack based threads

3 Performance of Stack-based Threads

This section discusses the optimizations we applied to our initial package as well as the fundamental limitations of a stack-based thread package.

3.1 Optimizations

The optimizations implemented in this phase are summarized in Figure 2. We discuss them in detail below. Many of these optimizations discussed in this section have been described in other papers; we include them for completeness.

Batch work: The first optimization is to remove work from the central queue in chunks. This reduces contention for the single central ready queue lock and (if threads are grouped in the queue according to the data that they touch) provides a primitive form of locality management. The disadvantage is that two linked list traversals are now necessary instead of one: the first to obtain threads, the second to run them. The Ultracomputer project used a similar mechanism [17].

Local queue: Central queues are a sequentializing bottleneck for the creation and execution of threads [3]. Furthermore, on machines that have a gross difference in speed between the CPU and memory, locality must be explicit for reasonable performance [25]. Local ready queues (each server owns a queue) eliminates contention for a central lock and allows thread placement to be based on locality (i.e., threads working on the same data can be explicitly placed on the same queue).

Disabled load-balance: This optimization actually reduces the complexity of the thread package. Due to the need to synchronize memory operations, on any machine where locality matters, such as the SGI Iris, locking is likely to be expensive as well. To eliminate locking overhead, load-balancing and locking queue operations for both RTC and barrier benchmarks are disabled (we describe a more realistic approach to reduce locking later in the paper). For unbalanced problems, load-balancing can be implemented by randomly assigning threads to queues.

Lazy save is a simple optimization: if a thread is exiting, there is no need to save any of its registers, and if a thread has not yet been run, there is no need to load anything more than its arguments, stack pointer, and return address. This optimization applies well to RTC threads since they complete without blocking. Since the bulk of barrier thread operations are blocking and restarting, they derive little benefit from this optimization.

Inexpensive index is an indulgence in bit-pushing: Dequeue and enqueue operations and some support functions are inlined and the size of each cell in each local queue is made to be a power of two (allowing indexing to be done by a single shift instead of multiplication or some number of shifts and adds). The effect of the added padding increases with the number of servers because of reduced contention. This optimization is simple, but without it a subtle performance bug is incurred when the size of frequently indexed queue structures is not a power of two.

Lazy stack is an optimization from [3]. At thread creation there is no need to immediately associate a thread descriptor with a stack. This association is necessary only at thread execution. If many threads complete without blocking, deferring stack allocation until thread execution requires that only two stacks are active (one stack for the exiting thread, another for the thread about to run). To support this optimization, however, the thread structure must be augmented with a pointer to thread code and any arguments that are used. Additionally, arguments must be copied into the thread description at creation time and then back onto the stack before execution. Furthermore, threads that have stacks must now be distinguished from those that do not. If many threads are blocked at any one time, this optimization will actually decrease performance. The benefit, however, is close to a two-fold improvement in creation time and introduces the possibility of a substantial decrease in memory consumption.

Live-stack takes advantage of the argument storage required by the previous optimization. Previously, when a thread exited and obtained the thread descriptor of another runnable thread, it would perform an optimized context-switch into it. However, because the exiting thread already had a "live" stack (stack pointer and program counter are already in place with valid values) the new thread could have been called directly, as a function. For our run-to-completion tests this optimization gives a threefold improvement in performance.

Optimization	**Create** $2.5x10^5$ **threads**	**Jacobi 100x100**		**Nop** $2.5x10^5$ **threads**		**Fib 24**		**Matrix 250x250**	
Central Queue	2.36	7.92	3.81	4.13	2.42	3.26	2.27	23.42	8.62
Batch Work	2.46	6.660	3.850	3.160	2.150	3.890	1.890	23.87	6.35
Local Queue	2.43	7.01	3.03	3.60	1.51	3.25	1.61	23.70	6.13
Disabled Load-Balance	1.68	5.75	2.72	2.51	1.33	3.27	1.56	23.63	6.19
Lazy Save	1.64	5.76	2.70	1.70	1.29	2.88	1.32	22.82	6.12
Inexpensive Index	1.55	5.42	1.63	1.60	0.70	2.79	0.97	22.89	5.85
Lazy Stack	0.82	5.99	1.94	1.53	0.50	3.19	1.02	23.33	5.97
Live Stack	0.82	5.99	2.10	0.46	0.18	2.91	0.95	22.38	5.78
Stateless	0.77	1.72	0.39	0.43	0.18	2.92	0.91	22.47	5.79

Figure 3: Cumulative effect of all optimizations (times in seconds). The two columns under each benchmark refer to the times for 1 and 4 processors; the single column under **Create** is the time necessary to create the indicated number of threads. The last row enumerates the times of a the initial stateless implementation; This is provided for reference only.

3.2 Discussion

Figure 3 summarizes the effects of the optimizations so far. The optimizations have reduced both thread creation and fork/join times by a factor of three, barrier iteration by up to a factor of two, and null execution overhead by an order of magnitude. Except for gains due to better locality, matrix does not improve by much since the amount of work per thread (a whole dot product computation) is large enough to mask most thread package overhead. The other applications profit more because of smaller computational tasks.

There are a number of problems associated with the thread model we have been discussing so far:

- High memory consumption. Each thread needs its own stack to create activation records and to store registers during context switching. Stacks are typically a multiple of the page size used by the operating system. Fine-grained applications can have thousands or millions of threads; for example, a thread can be created for each dot-product computation in matrix multiply. Using a typical page size of 4Kbyte and only ten thousand active threads easily consumes 40-Mbyte of memory. *Lazy allocation* may alleviate this problem, but does not solve it in general.

- Thread switching is expensive. The scheduler saves processor state on the stack of the blocking thread and restores the processor state of the resuming thread. As RISC machines evolve context-switching is becoming more costly. The processor state is increasing as processors have more registers. Additionally, storing and loading values from main memory is getting more expensive relative to CPU cycle time [27], [4].

- High Code complexity. To create, switch between, and terminate threads the thread package must explicitly manipulate thread stacks. While the code necessary to perform these tasks is short, it is notoriously difficult to get correct, as it has to deal with the idiosyncrasies of the underlying architecture. For example, on the SPARC, the code must ensure that there is always a valid user stackpointer on which hardware interrupts can be executed [20].

- Limited Portability. Because thread packages explicitly manipulate stacks and registers, they are architecture dependent. To port a thread package typically requires rewriting the most difficult part of the package using assembly instructions and requires knowing the conventions of the target machine. Given the increasing complexity of pipeline interactions and register saving conventions, this may be a time-consuming job; for example, the public domain Pthreads package that implements a draft of the POSIX standard contains 800 lines of SPARC assembly code [26].

4 Stateless threads

The problems discussed in the previous section can be solved using *stateless threads*, which are threads that do not support any blocking primitive requiring a context switch. At an operational level this means that all threads are represented solely by a pointer to a procedure and an argument block. Thread execution is accomplished by indirect calls; blocking is done via recursion. In this section we will present stateless threads, their semantics, optimizations that can be applied to them to increase performance

4.1 Semantics of Stateless Threads

On the surface, it seems that disallowing context switching makes the resultant package unusable. However, we found that once the restriction placed on allowable blocking operations was made explicit, a surprisingly large class of problems could be couched in terms of stateless threads. Additionally, most of the problems we have encountered that do not conform to this requirement can be easily expressed in terms of continuations and are hence still supportable.

We step through our three problem types to elucidate the semantics of stateless threads.

RTC threads are easily made stateless since they have no previous history, and they by definition will complete execution without blocking and thus do not need any capacity for saving state. Clearly, RTC threads can be implemented as procedure calls. Therefore, only a pointer to code and an argument vector is needed to represent and activate an RTC thread (or any stateless thread). Thread execution becomes a mere function call; thread creation is a handful of assignments. For example, the thread code for matrix addition (where there is one thread per point) is

```
/* A = B + C */
void add_point(int *aij, int bij, int cij) {
     *aij = bij + cij;
}
```

The thread takes three arguments, computes its result, and returns. In the context of stateless threads, thread execution is a function call. Thus thread exit is accomplished by simply returning to the caller. The only difference between RTC stateless threads and procedure calls is that a scheduler determines when and where the thread is run.

Barrier threads can be made stateless similarly by noticing that local state does not (usually) need to be preserved across iterations. Code with multiple barriers must be split into continuations. In our experience, this process is straightforward, and could be done by a compiler. A barrier thread is very similar to a RTC thread in implementation except that instead of a single invocation, there may be many. For example, to implement Jacobi iteration we create N^2 threads, and specify "sequential" code that is run between iterations to detect termination (and to allow us to reinitialize any state). To carry out the computation, the thread package completes all threads (as in its implementation of RTC threads) and then calls the following `user_code` to determine if the computation has completed. If it has not, all threads are run again, otherwise the calculation is halted. An implementation of this follows:

```
/* Jacobi iteration point --- average four neighbors and 'block' at barrier
 * barrier is implicit at the end of the code.  user code is used
 * to control the termination of the iterative process. */
void jacobi_point(int i, int j) {
        double diff;
        new_grid[i][j] = (old_grid[i-1][j] + other neighbors) / 4.0;
        diff = abs(old_grid[i][j] - new_grid[i][j]);
        /* if diff > maxdiff, atomically assign diff to maxdiff */
        atomic_asgn(diff,maxdiff);
}
/* user code to control termination of barrier iterations */
int user_code() {
        /* if our tolerance is satisfied, or too many iterations have
         * occured, signal termination */
        if(maxdiff < threshold || iterations++ > maxiterations)
              return DONE;
        /* otherwise reset the difference variable */
        maxdiff = 0.0;
        double **tmp = old; old = new; new = tmp;
        return !DONE;
}
```

Fork/join threads are more subtle; the general method used to make it stateless will be very useful later on. Typically, fork/join requires that a thread be capable of suspension (sleep) and subsequent wake-up. In short, threads must be able to save state.

How can we save state using stateless threads? Since threads are executed as subroutine calls, state will automatically be saved, by the compiler, at the callsite. In other words, if thread A invokes thread B, thread A's activation record will hold its state until B completes. A simple example should make how this abstraction can be applied to fork/join clear. The semantics of a join operation are that a parent thread waits for all of its children to complete, either by sleeping (as in traditional thread packages) or by having a signaling mechanism in place that indicates when all of its children have completed. The signaling mechanism we use is a child counter that is incremented on child creation and decremented on child completion. When this counter is zero, the parent is guaranteed that all of its children have completed. Thus to implement join without actual suspension we need only have the parent wait for its child counter to become zero.

```
/* 'sleep' until all children have completed' */
void join() {
     while( I have kids )
         run threads; /* execute other threads (possibly children) */
}
```

We have exchanged function calls for context-switches and recursion for large amounts of wasted stack space. This solution works because fork/join threads cannot have circularity: i.e., children are always created after their parents. As an example, consider a recursive Fibonacci calculation:

```
/* calculate Fibonacci number l - store result into stack allocated storage */
void fib(int n) {
     if(n <= 2)
          return 1;
     else
          return fib(n-1) + fib(n-2);
}
```

To parallelize this implementation all calls to `fib` are made into fork calls and, since fork returns its value asynchronously, storage is provided for the result. Finally, a call to join is done at the point at which we need the result of our fork calls:

```
/* calculate Fibonacci number l - store result into stack allocated storage */
void fib(int l, int *res) {
     if(l <= 2)
          *res = 1;
     else {
          int res1, res2;
          fork(fib, l-1, &res1);  /* fork two threads */
          fork(fib, l-2, &res2);
          join();                 /* wait for them to complete */
          *res = res1 + res2;
     }
}
```

Optimization	Description	Implication
Stateless	Stateless threads.	No stacks. Restricted thread model. Highly portable.
Bitpush	Optimized child counter decrement, increment. Two different thread structures (join, no join). Do not update global state.	Save storage and reduce locking.
Array	Use arrays and prune fork/join.	Save storage and create threads very cheaply. Large increase in complexity to manage arrays.
Cache	Execute threads back and forth for better caching. Unroll loops and inline creation.	Better caching and more efficient thread execution.
Load-Balance	Reintroduce load-balancing.	More effective support of uneven parallelism. Overhead in the non-load balancing case.

Figure 4: Overview of Optimizations Applied to Stateless Threads

4.2 Implementation Implications

There are four key benefits associated with stateless threads:

- No context switches.
- Thread mechanisms can be written in a high-level language. Because stack manipulations have been eradicated, no assembly language is required to implement a stateless thread package.
- Memory consumption has been lowered. Stateless threads use a stack *per process*, instead of per thread. Furthermore, this stack is managed by the system, not the package itself. Stack growth is thus dynamic and efficient. Additionally, the address space is not fragmented by the presence of many fixed-size thread stacks. This greatly reduces the possibility of thrashing or disrupting TLB management, both of which can easily occur when many stacks, each on a separate page, are touched.
- As discussed later in the paper, because of the particular semantics of stateless threads, situations that formerly required a context-switch (e.g., multithreading a lock) can now be done via recursion.

4.3 Optimizations

.23

The optimizations of this phase are summarized in Figure 4. Many of the optimizations given can also be applied to more traditional thread packages.

Bitpush is comprised of three bit-pushing optimizations.

Since threads that can be joined are distinguished at creation time from those that cannot (to avoid various bookkeeping costs), descriptors for joinable threads are now explicitly made distinct as well, reducing the storage consumption of non-joinable threads from 8 to 4 words.

The second bit-pushing optimization affects only RTC and barrier threads. Because we know where threads will exit and how they can block, we can make a host of simple optimizations because global, consistent versions of various thread data structures are not necessary. For example, to execute all threads in a queue, we just march down the thread list using pointer increments rather than a formal dequeue operation (which is necessary if a global pointer to the next runnable thread must be maintained). These optimizations give about a factor of two to three improvement in the null benchmark.

The final bit-pushing optimization consists of successive refinements to how fork/join child completion is recorded. In a naive scheme, the parent's child counter is atomically incremented at creation and atomically decremented at completion. To join, the server waits until its child counter is zero before proceeding. This counter can be separated into two variables, one non-atomically incremented on creation and decremented on completion of a local child by the parent, the other atomically incremented in the rare case when a child is remotely executed. The parent waits until these two counters are equal, at which point all children must have completed.

Array: The third optimization is to use arrays instead of queues for RTC and barrier threads. Using arrays for thread storage has a number of advantages:

- Space. No "next thread" pointer is needed. On the machine used for this paper this saves 20% of the space needed for each thread descriptor (20 vs. 16 bytes).
- Efficient creation. Allocation and enqueue of a thread descriptor requires a single condition check and pointer increment (to see if the array is full and to return a pointer to the next element). Creation cost is cut by 30%.
- Efficient load-balancing. Because threads are stored in arrays instead of linked lists, the cost to remove a chunk of work is little more than a pointer increment and a handful of conditionals. Load-balancing by removing chunks of work is inexpensive.
- Parallel creation. Many algorithms have a single producer of threads. If arrays are used, the producer can easily create threads on consumers' "queues" without locking. If a linked list was used instead, this might not be feasible, since removing the last element of a linked list when an unlocked enqueue is occurring can be difficult to implement correctly.
- Efficient storage management. Storage for thread descriptors is physically allocated and deallocated in chunks, instead of per thread block. This amortizes management costs at the expense of holding possibly needed storage.

The downside to an array representation is that extremely careful bookkeeping must be done if parallel creation, load-balancing and thread execution can occur at the same time. Of the approximately 1000 lines of code that comprise Filaments, about 400 deal with managing the array representation of thread descriptors.

A more dramatic (and simpler) optimization that is done for fork/join threads is to "prune" threads when sufficient parallelism is present. At thread creation, the server checks the number of runnable threads already on the ready queue, and if this number is greater than some threshold, the thread and all of its progeny are executed as recursive calls. This gives an order of magnitude improvement in performance of fork/join threads. WorkCrews is the first recorded thread package to use this optimization [30].

There are two subtleties to this otherwise straightforward optimization: the first is that in divide-and-conquer algorithms, the earliest threads will have the most work and so should be immediately accessible. Without the use of queue semantics (i.e., FIFO) little speedup will occur. This was first noted by [30].

The second is that, for simplicity, if a thread decides to prune a child, it prunes all of that child's progeny as well. An unfortunate result of this is that if other servers run out of work, there is no way to signal the pruning thread to create children instead of executing them directly. To ensure good performance, there should be provisions to ensure that the situation above does not occur frequently. A naive approach is to use an a-priori threshold chosen so that any server with that many threads is reasonably safe in pruning. The problem here is that if the threshold is very high, most of the benefits of pruning are lost and, conversely, even a high threshold is not guaranteed to be sufficient. A simple solution to this is to guide the pruning threshold using past behavior: When a server prunes, the number of elements is recorded, and if upon returning no threads have been taken, the threshold is lowered. If, however, work has been taken, more will likely be needed in the future and so the server raises the threshold. If all of the servers' threads are gone, the threshold is raised dramatically. The difficulty with such an approach is, of course, that the past may be no indicator of future events. A solution is to provide a mechanism for idle servers to interrupt the pruning of their more privileged brethren.

Cache: thread execution loops are unrolled and iterative threads are scheduled according to "most recently run", this provides good locality by running threads whose control blocks and data are most likely to be in a processor's cache. Additionally various procedures were inlined. When both the fork and the join mechanisms were inlined, the Fibonacci benchmark improved by a factor of two. The payoff of inlining is actually much greater than it should be: the vendor-supplied compiler eagerly allocates callee saved registers, even when there is a good chance that they will not be used. The upshot is that there can be 4-8 unnecessary loads and stores at every function entrance and exit. These memory references can be eliminated using "shrink-wrapping" [10].

Load-balance: The final optimization is to reintroduce load-balancing for RTC and barrier threads. The algorithm used to obtain work is a simple "scan and get" policy: an out of work

Optimization	Create 10^6 threads	Null $1x10^6$ threads		Jacobi 200x200		Fib 30	
Stateless	3.08	1.75	0.69	9.64	2.25	52.44	15.68
Bit-Push	2.63	0.92	0.25	8.21	2.05	37.93	12.19
Array	1.95	0.81	0.23	8.04	1.95	5.96	1.56
Cache	1.24	0.69	0.24	7.53	1.69	3.00	0.78
Load-Balance	1.18	0.70	0.22	7.63	1.78	3.00	0.77

Figure 5: Performance of stateless threads (times in seconds). The two columns under each benchmark refer to the time for 1 and 4 processors; the single column under **Create** is the time necessary to create the indicated number of threads. Sizes for all benchmarks have been increased for better measurement.

server scans other local run queues for work, and steals from the one with the most threads. In the final version, users are allowed to tailor load-balancing for their specific domain. Creation is sped up slightly because we are using the final Filaments package which requires that a pointer to a local run queue be specified by the user at thread creation. Because we the chunks of work removed at a time are roughly of size `(number of Filaments) / (number of servers)^2` instead of the size of a whole stateless thread array, the handling costs for removing threads are more noticeable.

4.4 Results

The results of the detailed optimizations are given in Figure 5. Creation overhead has been reduced by approximately a factor of three, as has null execution cost. Jacobi has improved by approximately 25% and Fibonacci by 25 times.

4.5 Implementation and Performance of Filaments

The performance of Filaments in relation to coarse-grained approaches is discussed in [24] and [18]. To test the overhead of Filaments relative to a coarse-grained implementation, pathological cases were examined. Jacobi iteration, which consists of very few operations, is within 11% of a coarse-grained implementation on a Sequent Symmetry, and within 40% on an SGI Iris. Considering that thread-based models have been found to be over twenty times slower than a coarse-grained application for this particular problem [23], [16], our performance is good. Fibonacci, which consists of a conditional and at most a single addition and two function calls is within a factor of two of a purely sequential implementation. For more reasonable fork/join programs (e.g. quicksort, adaptive quadrature, FFT) there is no difference at all.

We have found that the limiting effect on Filament's performance is not the overhead of creation and execution of threads, but the difficulty compilers have with inter-procedural optimizations. For example, floating-point constants in Jacobi iteration must be loaded for every filament, rather than being allocated on an inter-filament basis.

5 Exploiting Stateless Semantics

Multithreading has traditionally been a slow operation. For example, [21] cites times in the hundreds of instructions. This should not be surprising since the operations to switch between two threads T and T' and back (the sequence of operations required for multithreading) involves *at a minimum* the following operations:

1. Enqueue T on blocked queue.

2. Dequeue T' from run queue.

3. Switch from T to T'.

4. Dequeue T from blocked queue.

5. Switch from T' to T.

Total cost is two context-switches and three queue manipulations. This overhead significantly reduces the situations in which multithreading is profitable. The need for efficient multithreading is great enough that several machines provide hardware support. For example, the MIT J-machine can switch between a restricted thread class in 2-5 cycles [13]. The MIT Alewife machine, also supports very efficient context-switching but, again, only within a limited number of threads [1].

In this section we present a software solution to the multithreading problem that exploits the semantics of stateless threads to multithread in time comparable to a hardware solution, eliminating the need to go to an exotic machine. Furthermore, using stateless threads does not limit the number of threads that can be used to multithread.

Our solution uses the same observation used to implement the join operation: recursion can be used to temporarily save state if threads do not have circular dependencies. Blocking is accomplished by simply dequeuing a thread and calling it (these actions correspond to 1-3 in the traditional solution) when the thread returns (actions 4-5) the blocked thread resumes execution. The cost of multithreading has been reduced to a single dequeue and function call, or about 12-15 instructions. Two examples are provided to make this clear.

The first example is multithreading a lock. The approach is very simple: On failed lock acquisitions a thread dequeues and calls another thread. When the invoked thread returns, the thread attempts to reacquire the lock.

```
/* multithread a lock using recursion to save state on the 'blocked' threads
 * activation record */
void lock(lock_t lock) {
        /* while we cannot acquire lock, run threads */
        while(!grab_lock(lock)) {
                if( can dequeue thread )
```

```
                    run it;
          }
}
```

Context-switching costs and the enqueue/dequeue of the blocked thread are completely avoided. By dramatically reducing the costs of multithreading, applications can profitably exploit new levels of parallelism.

The second example shows how to implement a barrier in the middle of a stateless thread without breaking the thread up into continuations. Consider the code

```
void barrier_thread(args) {
     ...code...
     barrier(sequential_code);
     ...code...
}
```

To implement this primitive, each thread executes until it hits the barrier, at which point it attempts to dequeue another thread to run. When it cannot obtain more work, each server synchronizes with its peers (as in the old barrier implementation), then returns into each blocked thread and completes the code remaining after the barrier. The code required is, again, simple:

```
void barrier(code) {
     if( can dequeue thread )
          run it;
     else
          synchronize();
          if(server id == 0) code();
          synchronize();
}
```

The blocked threads form a recursive chain that synchronizes at the end and then propagates control back up the chain. This primitive has very attractive performance implications for a traditional thread package (assuming our constraints can be honored), namely instead of requiring that the thread synchronizing at the barrier incur the cost of blocking (i.e., context-switching, being enqueued on a blocked queue and subsequently being taken off), the thread package can now just recurse.

The notion of state saving when out-of-order resumption of blocked threads is not necessary allows a simple means of achieving very high levels of performance. This abstraction and the elimination of all machine language are the most fundamental contributions of Filaments.

6 Related Work

There is a wealth of related research on threads packages, some of which support fine-grained parallelism. The first step towards efficient parallelism was lightweight thread packages such as

Threads [14], Presto [6], uSystem [9], C Threads [11], uC++ [8] and Sun Lightweight Processes [28]. We will call these standard packages to distinguish them from thread packages that do not have a stack for each thread. The goal of standard packages is to provide the user with a natural thread abstraction, and many of the usual concurrent programming primitives; different packages provide different primitives. All of the above packages create a stack for each thread and support preemption to provide fairness. However, having stacks requires context-switching code, which is inherently machine dependent. Most of the above packages also use a central ready queue.

Standard thread packages, while providing the user with programming convenience, make it difficult to support fine-grained parallelism efficiently, because of the need to context-switch in and out of threads, the pollution of data space (specifically the cache and page tables) with stack memory, and the use of central queues. Finally, the generality of standard packages increases their size and overhead. Filaments is efficient because it has only one stack per server (processor), does not context switch, uses local ready queues and provides the minimal set of primitives needed for parallel implementations of single applications.

Several researchers have proposed ways to reduce the inefficiencies of standard thread packages. Anderson et al. [3] discusses the gain from using local ready queues, and [2] shows how to do user-level scheduling [2]. Markatos et al. [25] presents a thorough study of the tradeoffs between load balancing and locality in shared memory machines. Schoenberg and Hummel [19] explain how to avoid allocating a stack per thread and context switching in nested parallel `for` loops, which are a form of run-to-completion threads.

Thread packages that support finer-grained parallelism include the Uniform System, WorkCrews, TAM and Chores. The Uniform System [29], built for the BBN Butterfly, has several things in common with Filaments: there is no private stack per thread, no context switches and threads are not preemptable. The Uniform System's synchronous mode supports a simple form of barrier threads, and their finalization code is equivalent to our sequential code. However, barrier threads in Filaments are more powerful and they need be created only once. The Uniform System does not support fork/join threads. Another difference is that the Uniform System uses a central ready queue. In contrast, Filaments uses local ready queues; this minimizes lock overhead and enhances data locality. The Uniform System also employs task generators (a related collection of tasks)—and hence has essentially a coarse-grained programming model—whereas Filaments directly supports a fine-grained model. Because of these differences, the Uniform System cannot efficiently support thread-per-point decompositions for problems like Jacobi, as Lin [23] and others have noted. Furthermore, the model of parallelism is one of loop-based, not thread-based, parallelism. Specifying function-level parallelism in such a setting is cumbersome and inefficient.

WorkCrews [30] supports fork/join parallelism on small-scale, shared-memory multiprocessors. The package is implemented in Modula2+. WorkCrews introduced the concept of pruning and of ordering queues to favor larger threads. Filaments has borrowed these ideas in its implementation of fork/join threads.

TAM [12] is a compiler-controlled threaded abstract machine. It evolved from graph-based execution models for dataflow languages and provides a bridge between such models and the control flow models typically employed by standard multiprocessors. TAM programs are specified in the assembly language of an abstract machine, which is in turn compiled by TAM into machine code. The scheduling of threads and the management of memory storage resources is also exposed. Consequently, TAM requires a heavier investment in compiler technology than Filaments, since user code must first be compiled to TAM assembly language and then compiled by a TAM compiler to machine instructions. In contrast, Filaments can easily be used by a programmer directly. We believe that removing the need for sophisticated compilers is an important advantage of Filaments, in that it allows Filaments to be easily ported to new machines with minimal effort.

A TAM thread is a sequence of instructions which executes from beginning to end without suspension or branching [12]. In contrast, Filaments can include any number of control flow changes and blocking primitives as long as none necessitate a context-switch. This weaker constraint allows Filaments to have larger tasks and thus the potential for more efficiency. Furthermore, while the fine-grained nature of the TAM machine allows very exact control of what executes when, there appears to be a noticeable cost in the number of control instructions executed for each instruction of useful work. For example, an implementation of quick-sort has 60.3 percent of its instructions devoted to "control" [12]. To be fair, this is a result of the TAM model being more ambitious than Filaments in that it is intended for use in message-based parallel computers and allows explicit management of multiphase transactions by the runtime system.

Chores [16], which runs on top of Presto on a Sequent Symmetry, is another threads package that is similar to Filaments. Chores, extends the Uniform System model by coalescing iterations into large tasks (in essence deriving coarse-grained parallelism from a fine-grained specification). Additionally, Chores provides support for fork/join parallelism. Like the Uniform System, Chores uses a central ready queue, but servers take jobs in chunks. This amortizes the lock overhead of the central ready queue. Chores is similar to Filaments in the same way the Uniform System is: there are no private stacks per thread, no context switches, and no preemption. Chores is more flexible than Filaments because user threads are run on top of systems threads that have a stack, which permits blocking (to be fair, the same effect could be achieved in Filaments by having servers execute on top of traditional threads). However, Filaments directly supports fine-grained parallelism, such as Jacobi iteration with a thread per point. Chores requires preprocessor support and the use of task generators in order to cluster fine-grained tasks into coarse-grain units. Two other drawbacks of the Chores approach are that extreme irregularity may make the coarse-grained tasks perform poorly, and the use of traditional threads (specifically Presto [6]) to implement a given chore makes fine-grained chores inefficient.

Finally, the concept of a thread barrier requiring time proportional to the number of servers and not the number of threads was developed independently David Keppel [22]. The main difference is that there is no notion of sequential code.

7 Conclusion

We have discussed, in detail, a series of optimizations that can be done to increase creation and execution efficiencies of three thread classes. An important part of this transition is the use of stateless threads, the advantages of which are:

- No context-switching. Context-switching performance, because of larger register sets and slow memory speeds, is not scaling with integer CPU performance [27], [4]. By using stateless threads, this bottleneck is avoided.
- Low memory consumption. By using one system-managed stack, stateless threads consume far less memory than a traditional thread package. This results in better caching and TLB performance.
- Portability. Because stack manipulations are eliminated, no machine language is needed to create, manage and execute threads. This allows the entire package to be implemented in a high-level language, which makes it highly portable and easily maintainable.

Compared to our initial package, the end result of our optimizations is an order of magnitude increase in creation efficiency and up to 40 fold improvement in execution times. The final package can create threads at an average cost of 12 instructions and can run them in approximately 6.

8 Acknowledgements

The author would like to thank the following people at University of Arizona: David Lowenthal for many discussions and his thorough testing of Filaments' performance, Vince Freeh for his discussions about fork/join and most especially, Greg Andrews, without whose support the author would not have begun the research leading to Filaments or, possibly, any research at all.

The author would like to thank the following people at M.I.T.: Wilson Hsieh, Kevin Lew, and Deborah Wallach for their thorough reading and comments. And especially Professor Frans Kaashoek, who spent many days helping the author write this paper.

References

[1] Agarwal et al. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS TM-454, M.I.T., 1991.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the ThirteenthACM Symposium on Operating Systems Principles*, pages 95–109, 91.

[3] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Transactions on Computers 38*, 12:1631–1644, December 1989.

[4] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. The interaction of architecture and operating system design. In *Proc. Fourth International Conference on ASPLOS*, 1991.

[5] G. R. Andrews et al. An overview of the SR language and implementation. *ACM Transactions Programming Language System*, 10:51–86, January 1988.

[6] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: a system for object-oriented parallel programming. *Software—Practice and Experience*, 18:713–732, August 1988.

[7] B.N. Bershad and M.J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, CMU, 1991.

[8] Peter A. Buhr, Glen Ditchfield, R. A. Stroobosscher, and B. M. Younger. uC++: concurrency in the object oriented language C++. *Software—Practice and Experience*, 22:137–172, February 1992.

[9] Peter A. Buhr and R. A. Stroobosscher. The uSystem: providing light-weight concurrency on shared memory multiprocessor computers running unix. *Software—Practice and Experience*, 20:929–964, September 1990.

[10] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, June 1988.

[11] Eric C. Cooper and Richard P. Draves. C threads. Internal note of the Mach research project, September 11 1990.

[12] David E. Culler and et al. TAM—a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, 1993.

[13] William J. Dally et al. The J-Machine: A fine-grain concurrent computer. In G.X. Ritter, editor, *Proceedings of the IFIP Congress*, pages 1147–1153, 1989.

[14] Thomas W. Doeppner. Threads: a system for the support of concurrent programming. Technical Report TR CS-87-11, Brown, 1987.

[15] D. Dwarkadas, P. Keleher, A. L. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *The 20th Annual International Symposium on Computer Architecture*, pages 144–155, 1993.

[16] Derek L. Eager and John Zahorjan. Chores: enhanced run-time support for shared-memory parallel computing. *Trans. on Computer Systems*, 11:1–32, February 1993.

[17] J. Edler, A. Gottlieb, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, M. Snir, P.J. Teller, and J. Wilson. Issues related to MIMD shared-memory computers: the NYU Ultracomputer approach. In *Proc. Twelfth Annual International Symposium on Computer Architecture*, 1985.

[18] Dawson R. Engler, Greg R. Andrews, and David K. Lowenthal. Filaments: Efficient support for fine-grain parallelism. In preparation.

[19] S. F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journal of Research and Development*, 35:743–765, September 1991.

[20] SPARC International. *The SPARC Architecture Manual.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[21] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the ThirteenthACM Symposium on Operating Systems Principles*, pages 41–55, 91.

[22] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE93-05-06, The Univ. of Washington, 1993.

[23] Calvin Lin and Lawrence Snyder. A comparison of programming models for shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 163–170, 1990.

[24] David K. Lowenthal and Dawson R. Engler. Performance experiments for the filaments package. Technical Report TR 93-26, The Univ. of Arizona, 1993.

[25] E. P. Markatos and T. J. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessors. In *Proc. 1992 International Conference on Parallel Processing*, pages 258–267, August 1992.

[26] F. Mueller. A library implementation of Posix threads under UNIX. In *Proc. USENIX Conference Winter'93*, pages 29–41, 1993.

[27] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. Summer Usenix*, pages 247–256, June 1990.

[28] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX*, 1992.

[29] Robert H. Thomas and Will Crowther. The Uniform System: an approach to runtime support for large scale shared memory parallel processors. In *Proceedings of the 1988 Conference on Parallel Processing*, pages 245–254, August 1988.

[30] M. Vandevoorde and E. Roberts. WorkCrews: an abstraction for controlling parallelism. *Int. Journal of Parallel Programming*, 17:347–366, August 1988.

A Appendix A

```
/* Header file for initial, traditional thread package */

/* Initialize package. */
sr_initialize();

/* Alter default stack size. */
sr_chg_stack_size(size);

/* Create a thread. */
sr_wire(void (*code)(), any arg1, any arg2, void *arg3);

/* Start servers, returns when all threads have completed. */
```

```
sr_start(void (*sequential_code)());

/* Terminate a thread. */
sr_exit();

/* Create a thread that can be joined. */
sr_fork(void (*code)(), any arg1, any arg2, void *arg3);

/* Join all outstanding threads. */
sr_join();

/* Barrier synchronize.  The package calls the user supplied sequential_code
 * to detect termination of barrier threads.  */
sr_barrier();

/* Allocate and initialize a lock. */
MULTI_ALLOC_LOCK(lock_t lock);

/* Lock. */
MULTI_LOCK(lock_t lock);

/* Unlock. */
MULTI_UNLOCK(lock_t lock);
```

B Appendix B

```
/* Filaments header file */

/* Start servers. */
f_start();

/* Allocate a queue, indicate whether it will be barrier, or run-to-completion.
 * Sequential code to run between iterations (or at completion) is provided */
f_qalloc(int (*sequential_code)());

/* Fork a thread. */
f_fork(void (*code)(), any arg1, any arg2, void *arg3);

/* Join all outstanding threads. */
f_join();

/* Create a thread on the given queue on the server 'pid's entry. */
f_thread(f_queue *q, int pid, void (*code)(), any arg1, any arg2, void *arg3);

/* Initialize package. */
f_initialize(nservers);
```