A Comparative Analysis of Fine-Grain Threads Packages[*]

Gregory W. Price and David K. Lowenthal
Department of Computer Science
The University of Georgia
Athens, GA 30602
Email: {price,dkl}@cs.uga.edu

May 30, 2001

Abstract

The rising availability of multiprocessing platforms has increased the importance of providing programming models that allow users to express parallelism simply, portably, and efficiently. One popular way to write parallel programs is to use *threads* for concurrent sections of code. User-level threads packages allow programmers to implement multithreaded programs in which thread creation, thread management, and thread synchronization are relatively inexpensive.

Fine-grain programs are multithreaded programs in which the work is divided into a large number of threads, where each thread contains a relatively small amount of work. The potential benefit of large numbers of threads include easier load balancing, better scalability, greater potential for overlapping communication and computation, and improved platform-independence. However, fine-grain programs are largely considered inefficient due to the overheads involved in managing numerous threads. In this paper, we survey several thread packages that take different approaches to the problem of efficiently supporting the creation and management of large numbers of fine-grain threads. Each package is compared based on its level of support of the general thread model as well as its performance on a set of fine-grain parallel programs. We find that while the thread packages we tested may support medium-grain parallelism efficiently, they do not always support fine-grain parallelism. Although no package supports fine-grain parallelism *and* a general thread model, we believe that it can potentially be done with help from the compiler.

1 Introduction

A *process* is an abstraction of a physical processor. It is used to make systems programming easier to design and implement, both inside and outside of the operating system (OS) kernel. Typically, a process consists of the CPU state, the kernel stack, a working directory, open file descriptors, a signal table, the signal mask, the user id, the group id, and a memory map. The CPU state includes a stack pointer (SP), which points to the current activation frame on the stack, the program counter (PC), which references the current instruction, and the remaining registers that store other global and local data.

Concurrency is achieved by allowing several independent processes to exist at the same time. They are scheduled by the kernel, one at a time, to run on the CPU. As a result, if a process is waiting to access the disk or other I/O device, another process can be run while the first is idle. This model extends to a multiprocessor, where each processor can run a process simultaneously.

[*]This research was supported by National Science Foundation Grant CCR-9733063.

Unfortunately, switching from one process to another introduces a large amount of overhead. The CPU state and the other process information must be stored, the TLB is typically flushed and reloaded, and the next scheduled process must restore its own CPU state and other information before executing. Communication between processes is also cumbersome.

A *thread* is a lighter-weight process developed to overcome many of these shortcomings. Multiple threads can exist within a process and will share the memory map, the file descriptors, code, and global data. The threads themselves consist only of a program counter, stack pointer, stack, general purpose registers, and a small amount of additional thread management information. The lower overhead of threads makes it much easier to achieve speedup in parallel programs in which a thread is assigned to each logically independent unit of work. On the other hand, due to the larger overhead, the use of multiple processes can result in speedup only if each process performs a significant amount of computation.

Many operating systems are implemented using *kernel threads*, which are created by the OS to perform various internal tasks such as scheduling, running user programs, and handling page faults. To protect the kernel from errant user programs, such tasks are executed in kernel space, and user programs that access the disk or otherwise interact with the kernel are required to make a system call. Such a call causes a switch to kernel space, execution of the desired task, and a return to user space.

User threads, on the other hand, are those that are created by a user to take advantage of concurrency in the code. User threads have the advantage that the kernel need not be involved in creation, scheduling, and context switching. This avoids excessive user-kernel boundary crossings. Such crossings must be minimized in order to achieve good performance when tens or hundreds of thousands of threads are created. Because this is often the case in very fine-grain parallel programs (see below), in this paper we focus exclusively on user-level threads packages.

The purpose of this paper is to evaluate four user-level threads packages: Cilk [BJK$^+$95, FLR98], Filaments [FLA94, LFA96], Lazy Threads [GSC96] and StackThreads/MP [TTY96, TTY99]. Each claims to provide support for large numbers of efficient, fine-grain threads. We compare these packages in two ways. First, we examine how closely each package supports the thread model that is generally accepted by programmers of multithreaded applications. Second, we measure the performance of each package on benchmarks that create huge numbers of threads, where each thread performs little work. We examine both the overhead relative to a sequential (single-threaded) program as well as speedup when a second processor is added. It is important to note that we are *not* testing these packages on a range of fine-, medium-, and coarse-grain programs. We are focusing exclusively on how well each package supports fine-grain programs.

We found that each package excels in at least one dimension, but also has some drawbacks. The integration of Lazy Threads with a compiler allows it to support the general thread model and provide efficient fine-grain parallelism, but the dependence on the compiler hampers portability. StackThreads/MP removes this compiler dependence, but cannot efficiently handle (1) iterative parallel programs and (2) threads with very little work. Filaments provides efficiency in fine-grain programs across multiple application domains, but does not support a general threads model. Finally, Cilk is highly scalable for medium grain applications, but is inefficient when the granularity is much smaller. It also supports only recursive parallelism efficiently.

The remainder of this paper is organized as follows. Section 2 describes our general thread model, and Section 3 discusses drawbacks and potential benefits of fine-grain threads. Section 4 introduces and examines four user-level threads packages that support fine-grain threads. Section 5 evaluates the performance of each package with a set of benchmarks that use fine-grain parallelism. Finally, Section 6 presents a summary of features for each package and concludes.

2 General Thread Model

As discussed in the previous section, in order to examine each threads package relative to the generally accepted threads model, we must define such a model. Our model is patterned after two popular threads packages: Solaris Threads[PKB$^+$91] and PThreads[Ber98]. It contains the following elements.

> *Creation*: Threads can be created (often called *forked*) at any point during program execution. The user specifies a procedure that the thread is to execute. Creation involves allocating space for the thread descriptor and some amount of space for a private stack. The thread descriptor holds the thread id, a pointer to its parent, a counter and/or pointers for any children created, and other thread specific data,
>
> *Destruction*: Upon completion, threads are deallocated, return values (if any) are stored, and the parent of the terminating thread is notified of the status of the child.
>
> *Independence*: A thread can continue after its parent terminates. A multi-threaded process will not terminate until all threads have completed[1].
>
> *Blocking*: Threads can voluntarily be suspended at any time (e.g., block) or can be involuntarily suspended by the scheduler so that another waiting thread can be run. A blocked thread can sleep for a specific amount of time or until a given condition is met. Threads typically block to wait for I/O, for another thread to finish (often called *join*), or for access to a lock or other form of synchronization (see below). Blocked threads may be resumed by the scheduler at any time after the condition is met. Note that in a typical fork/join model, a thread forks n threads and then joins on some or all of them.
>
> *Synchronization*: Since threads can share code and global data, a mechanism is necessary to prevent multiple threads from accessing the same location at about the same time, where at least one thread writes to that location. This *race condition* will cause nondeterministic (and sometimes fatal) results from one execution to the next depending on how the threads are scheduled. Race conditions can be circumvented by allowing only one thread to access global data at a time using *locks* inserted by the programmer around these *critical sections* of code. Other constructs such as *semaphores* and *condition variables* [And00] can be used to ensure that threads access shared data in the proper order.

3 Fine-Grain Parallelism

One way to implement a parallel program for a multiprocessor machine with p processors is to divide a program into p units and create one thread per unit. The size of the division is referred to as the *granularity*. The left-hand side of Figure 1 shows a *coarse-grain* division; the advantage to such divisions is that only the exact number of threads needed are created; consequently, any thread overheads (see below) are minimized. The disadvantages to a coarse granularity are (1) the programmer is required to divide the work into p units, which limits the portability of the code, and (2) if the units do not contain an equal amount of work or the processors are of different speeds, one processor may become idle while another completes a majority of the work, which reduces scalability and makes balancing the load between processors more difficult.

The other extreme is a *fine-grain* program, in which each unit encompasses a small, independent part of the computation. (See the right-hand side of Figure 1.) With traditional threads packages, the overhead for creating the threads far outweighs the benefits gained from creating massive parallelism. On the other

[1]Some newer, thread-friendly, operating systems will terminate the process if no threads are runnable (ie, two threads waiting for each other to complete).

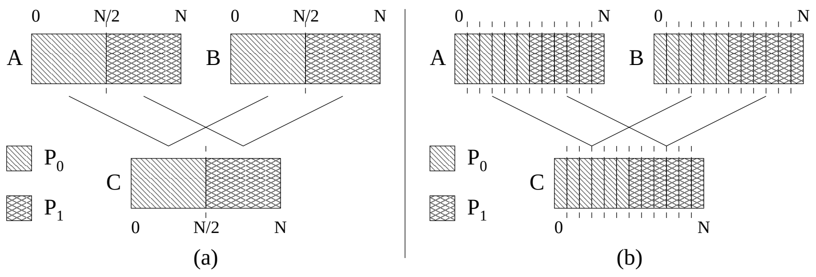


Figure 1: Three vectors are used to calculate $C = A + B$. In (a), the vectors are divided into two parts (coarse-grained) with one thread per division. Each processor will execute one thread and complete half of the work. In (b), the vectors are divided into many parts (fine-grained) with each part being controlled by a single thread. Each processor executes a number of threads but still completes half of the work.

hand, automatic code generation is simpler since parallelism can be created based on loops or recursive procedure calls instead of clustering work into some number of larger units (usually based on the number of processors). Also, the programs are more scalable since the number of threads created will usually be much more than the number of processors on the system. As a result, load balancing is simplified; when one processor completes its work, another waiting thread can be run. Note that a particular fine-grain threads package can implement load balancing through (1) work stealing, where idle processors steal threads from busy processors, (2) work sharing, where busy processors donate threads to idle processors, or (3) explicit scheduling on another processor. Section 4 has more details on the load balancing techniques of the thread packages.

Iterative and divide-and-conquer applications alike can be ideal for fine-grain parallelism. The vector sum in Figure 1 shows an iterative application; with fine-grain parallelism, the concurrency is expressed in terms of the application, not the architecture. Divide-and-conquer programs can use a fine-grain thread to execute each independent recursive call in parallel. There is often no simple analogous coarse-grain program (a coarse-grain, bag-of-tasks algorithm [And00] approximates the fine-grain program but is significantly more difficult to code).

Fine-grain parallelism is not without significant potential costs, however. Each thread-related action introduces some small amount of overhead to the program in the following ways:

1. The cost of creating, scheduling, and terminating a thread introduces additional overhead compared to calling the procedure directly.

2. When threads access the same data, additional code is required to prevent race conditions and other unexpected results.

3. Since threads can be scheduled in any order, some efficiency is lost because most compilers are designed to optimize loops. When threads are created for individual loop iterations, as is often the case in parallel programs, these optimizations are lost.

4 Fine-Grain Threads Packages

A number of research groups have developed methods for the efficient support of fine-grain threads. Cilk ([BJK$^+$95, FLR98]) uses a theoretically efficient scheduler and a restricted thread model to efficiently manage threads. Concert[PKZA, CKP93] is a full compiler designed with heavy optimizations for efficiency of multithreaded programs. Filaments ([FLA94, LFA96]) also restricts the thread model to reduce the overhead involved with creating a large number of threads. Lazy Threads [GSC96] is built into the compiler and uses an efficient method of stack management that can support fine-grain threads. Finally, StackThreads/MP ([TTY96, TTY99]) attempts to reduce the cost of a thread to that of a standard procedure call. Each of these packages (with the exception of Concert[2]) will be discussed in detail in this Section, and Section 5 evaluates the performance of these packages.

4.1 Cilk

Cilk was developed at the Massachusetts Institute of Technology and has been an ongoing project since 1992. Cilk-5.2, the current version, comes with a compiler, runtime system, race detection tool, source code, and documentation, which can be found at the Cilk Project web site http://supertech.lcs.mit.edu/cilk/index.html.

Cilk currently runs on Sparc Solaris, Linux PC, SGI/IRIX, and Alpha/OSF, and requires the GNU C compiler (`gcc-2.7.x` is recommended). Currently, Cilk supports SMPs, and a recent release supports distributed machines as well.

4.1.1 Cilk API

The original version of Cilk encouraged a continuation passing style of programming, which is different from typical fork-join semantics in that a continuation (a second thread) must be created. Return values from child threads are then passed to the continuation. In this style, parent threads cannot simply block and wait for their children to finish. It forces programmers to drastically restructure their applications to make use of these continuations. However, the most recent version employs an API that helps make programming easier. By using three simple keywords, a C-program can be converted into an equivalent Cilk program with minimal effort. A simple example of this can be seen in the Fibonacci example in Figure 2.

In the Fibonacci program (Figure 2), the results of `fib(n-1)` and `fib(n-2)` can be computed in parallel using fork and join. The keyword **cilk** indicates that the function is potentially parallel, **spawn** is the equivalent of the traditional fork that indicates the creation of parallelism, and **sync** blocks until all children spawned in the current scope have completed. By removing the keywords, the Cilk program can also be compiled and run sequentially by a traditional C compiler. For a Cilk program, the `cilk2c` compiler uses `lex` to convert these keywords into a parallel C program that can be compiled by `gcc`.

Finally, Cilk provides routines for synchronization through the use of traditional locks in the functions `Cilk_lock()` and `Cilk_unlock()` which access a built-in *Cilk_lockvar* type. Calling `Cilk_lock()` will block the current thread if the lock is already held until the lock is released.

4.1.2 Cilk Model and Implementation

Cilk employs a unique mechanism for efficient threads. Each Cilk thread is a non-blocking unit of work that can be executed in parallel with other spawned threads. For a given state of a program, a call graph can be constructed that indicates the dependencies, including the continuations necessary to follow Cilk's non-blocking thread model. Figure 3 shows one state of the call graph for the Fibonacci program. In this graph, horizontal lines represent continuations or successor threads, downward diagonal lines represent spawned threads, and upward diagonal lines show the dependencies between threads. In other words, threads `B`, `B′` and `C` can not begin execution until they are created by thread `A`, and thread `C` can not run until `A`, `B` and `B′` are finished.

This example demonstrates two important things about the Cilk model. First, since Cilk threads cannot block, the simplest way to implement fork-join semantics supported by the traditional thread model is to spawn a continuation thread (in this case, thread `C`) to run only after thread `B` and `B′` have completed.

[2]Unfortunately, Concert was not implemented for the test platform used in this paper and is no longer under development.

```
CILK int fib (int n) {
  if (n < 2)
    return 1;
  else {
    x = SPAWN fib(n-1);
    y = SPAWN fib(n-2);
    SYNC;
    return (x+y);
  }
}

CILK int main (void) {
  int total = SPAWN fib(20);
  SYNC;
  printf ("Fibonacci of 20 = %d\n", total);
  return 0;
}
```

Figure 2: An example Fibonacci program implemented using Cilk. Keywords specific to Cilk are capitalized.

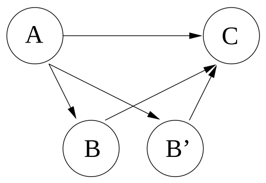


Figure 3: `A` spawns `B` and `B′` to calculate `fib(n-1)` and `fib(n-2)` respectively. `C` must wait until both `B` and `B′` have returned, because `C` must use the values computed by `B` and `B′` to calculate the sum.

Fortunately, the `cilk2c` compiler does this implicitly by translating the **spawn** and **sync** keywords mentioned above into an equivalent Cilk program. Second, all thread calls in Cilk can be diagrammed in this way, producing a directed acyclic graph that can be used to algorithmically analyze and prove the efficiency and correctness of the Cilk's work-stealing scheduler, which is based on a modified Dijkstra algorithm for mutual exclusion [Dij65].

Figure 4 demonstrates how wait queues can be structured to allow efficient work stealing. When a processor is idle (has no work to do), the Cilk scheduler first chooses a processor `p` at random and then selects a free thread from the highest level queue in `p`'s call tree. By stealing the highest-level thread possible (and consequently all of its children and successor threads), the thief is receiving a significant amount of work and more efficiently balancing the load. If, by contrast, the lowest level thread were selected, only one thread would be stolen. The thief would then complete the work and have nothing more to do, forcing it to make multiple steals instead of just one. Since stealing introduces a relatively large amount of overhead, multiple steals should be avoided. In a recursive program, this algorithm works well, as each high-level thread is more likely to have more descendants, and consequently, more work.

4.1.3 Advantages and Disadvantages of Cilk

Perhaps the most notable advantage of the Cilk package is the near-perfect speedup achieved as compared to the single processor version of the Cilk program. Doubling the number of processors doubles the per-

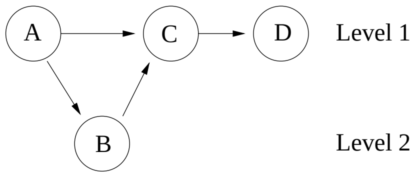


Figure 4: Structuring a queue at each level of thread creation aids in an efficient work stealing algorithm.

formance of the Cilk application. It is important to note that the near-perfect speedup is not speedup as compared to the sequential program, but as it compares to the single processor version of the Cilk program (see Section 4 for details).

As with any threads package, there is overhead introduced in a one-processor (single-threaded) Cilk version of an application. With Cilk, this is especially significant; Cilk one-processor programs always run much slower than their sequential counterparts in our tests. Unfortunately for Cilk, some applications, such as the Fibonacci program, can be as much as *nine* times slower than a sequential version of the same program. Another notable disadvantage of Cilk is that it relies on divide-and-conquer algorithms to produce efficient speedup through the use of the load balancing scheduler (see section 3.1.2).

Consider the call tree for an iterative program such as Jacobi iteration example in Figure 5. Each thread created would be a leaf node in the tree, and if stolen, only a small amount of work is migrated. Consequently, many more steal attempts are required in order to balance the load on the system, which in turn results in a significant decrease in performance of the parallel program. Transforming this algorithm into a divide-and-conquer style algorithm in Figure 6 will produce more efficient load balancing and better speedup as a result, but the transformation may not be a trivial task (for example, with LU decomposition, it is not). Rewriting iterative applications to take advantage of the recursive nature of Cilk is unfortunately left to the burden of the programmer. As a side effect of this transformation, many compiler loop optimizations are also lost.

4.2 Filaments

Filaments was originally developed at the University of Arizona in 1993 but is currently being maintained at the University of Georgia. Filaments contains the runtime system, source code and documentation, and is available for download from http://www.cs.uga.edu/˜dkl/filaments/dist.html (a 516K .tar.gz file).

The current version runs on Sparc/Solaris, X86/Solaris, X86/Linux, and SGI/IRIX. Filaments supports either shared memory machines or distributed memory systems – mixed platforms are not supported[3].

4.2.1 Filaments API

The Filaments library includes a number of macros and procedure calls to aid the programmer in writing Filaments programs. Routines exist to initialize the package, allocate and control memory, and to create `filaments` (lightweight threads). Fork-join and iterative algorithms use separate library routines, which makes the package more efficient for each type of program. In the Fibonacci program in Figure 7, the recursive `fib()` procedure is replaced with a Filaments header `filDeclareFJHeader(...)`, which has as parameters the return type, the name of the function, the type of the parameter, the parameter name, and the body of the function. The function `filCreateFJFilament(...)` indicates the creation of parallelism (fork) and `filJoinFJFilament()` synchronizes the forked threads (join).

In the body of the main function of Figure 7, the package initialization routines and other miscellaneous functions are invoked. Function `filInitFilaments()` initializes the runtime system and `filSetK-`

[3]Current research in Filaments is working on support for mixed systems.

```
cilk void update(int i, int j) {
  double temp;
  new[i][j] = (old[i][j-1] + old[i][j+1] + old[i-1][j] + old[i+1][j]) * 0.25;
  temp = (old[i][j] - new[i][j]);
  if (temp < 0)
    temp = -temp;
  Cilk_lock(mutex);
  if (md < temp)
    md = temp;
  Cilk_unlock(mutex);
}

cilk int main (int argc, char* argv[]) {
  double** temp;
  n = atoi(argv[1]);
  maxiters = atoi(argv[2]);
  Cilk_lock_init(mutex);

  /* Not Shown: Allocate and Initialize Matrices */

  while (numiters < maxiters) {
    md = 0.0;
    for (i = 1; i < n-1; i++)
      for (j = 1; j < n-1; j++)
        spawn update(i, j);
    sync;
    temp = old; old = new; new = temp;
    numiters++;
  }
  printf ("Done.  md: %f\n\n", md);
  return 0;
}
```

Figure 5: A simple Jacobi iteration example in Cilk.

```
cilk double updateR(int startI, int stopI, int startJ, int stopJ) {
  double t1, t2, t3, t4;
  int i, j, midpt;
  double maxdiff = 0.0;

  if ((stopI - startI) < 2) {
    maxdiff = 0.0;
    for (i = startI; i <= stopI; i++)
      for (j = startJ; j <= stopJ; j++) {
        new1[i][j] = (old1[i-1][j] + old1[i+1][j] + old1[i][j-1] + old1[i][j+1]) * 0.25;
        if ((t1 = new1[i][j]-old1[i][j]) < 0)
          t1 = -t1;
        if (t1 > maxdiff)
          maxdiff = t1;
      }
    return maxdiff;
  } else {
    midpt = (stopI-startI)/2;
    t1 = spawn updateR(startI, startI+midpt, startJ, startJ+midpt);
    t2 = spawn updateR(startI, startI+midpt, startJ+midpt+1, stopJ);
    t3 = spawn updateR(startI+midpt+1, stopI, startJ, startJ+midpt);
    t4 = spawn updateR(startI+midpt+1, stopI, startJ+midpt+1, stopJ);
    sync;
    /* Not Shown: Set maxdiff to the Greatest of (t1, t2, t3, t4) */
    return maxdiff;
  }
}

cilk int main(int argc, char **argv)
{
  ...
  while (numiters < maxiters)
  {
    md = 0.0;
    md = spawn updateR(1, n-2, 1, n-2);
    sync;
    temp = old; old = new; new1 = temp;
    numiters++;
  }
  ...
}
```

Figure 6: A recursive implementation of Jacobi iteration.

```
filDeclareFJHeader(int, fib, int, n,
{
  int r1;
  int r2;

  if (n <= 2)
    return 1;
  else  {
    filCreateFJFilament(fib, r1, n-1);
    filCreateFJFilament(fib, r2, n-2);

    filJoinFJFilament();

    return r1+r2;
  }
})

int main (int argc, char *argv[]) {
  ...
  filInitFilaments();
  filSetKernel(F_FJ_KERNEL);
  filEnableLoadBalance();
  filSetPrune(...);

  filCreateFJFilament1(fib, answer, n);

  filStartFilaments();
  ...
}
```

Figure 7: An example Fibonacci Sequence program implemented using Filaments.

`ernel()` chooses between fork-join or iterative filaments. With fork-join filaments, `filEnableLoad-Balance()` allows work-stealing to occur if a processor is idle[4] and `filSetPrune()` sets the pruning threshold[5]. Finally, the package is initialized and the first thread is run when `filStartFilaments()` is called, and the return of this function serves as an implicit join implying that the initial forked filament has completed.

An iterative algorithm that uses the Filaments library can be seen in the Jacobi iteration in Figure 8. Similar to fork-join Filaments, a macro surrounds the `update()` function with two integer arguments `i` and `j`, but in this case there is no return argument. Next, the function `filUpdateRedVar()` updates a special shared variable called a *reduction* discussed in section 3.2.2.

In function `main()`, after the initialization of memory, the reduction variable and the package itself, filaments are divided among separate processor *pools* for locality. Each processor runs filaments from its own pool until that pool is either empty or blocked due to a remote data reference[6]. A *phase* is a collection of pools and in this case, it represents one complete Jacobi iteration. The number of iterations run is determined by the reduction variable `maxdiff`.

[4]Dynamic load balancing is not performed when executing iterative Filaments; the work must be divided up by the programmer, a compiler such as SUIF [HAA$^+$96], or a run-time system such as Adapt [HL00].

[5]See section 3.2.2 Filaments Model and Implementation for details.

[6]A remote reference occurs only in a distributed environment when accessed data is on another node.

```
filDeclareIterHeader(update, int, i, int, j,
{
  double temp;
  new[i][j] = (old[i-1][j] + old[i+1][j] + old[i][j-1] + old[i][j+1]) * 0.25;
  if ((temp = new[i][j]-old[i][j]) < 0)
    temp = -temp;
  filUpdateRedVar(maxdiff, temp, <);
})

int main(int argc, char *argv[]) {
  int n;                                    /* Initialized to matrix size */
  ...

  filResetReduction(filGetRedName(maxdiff), R_MAX, T_DOUBLE);

  filInitFilaments();
  filSetKernel(F_ITERATIVE_KERNEL);

  /* Create a phase for each iteration in which update() is called */
  phase = filCreatePhase(2, update, checkConvergence, 0);

  /* Create a pool (queue) for each processor */
  pool = filCreatePool(phase, (n-1)*(n-1));
  for (i = 1; i < n-1; i++)
    for (j = 1; j < n-1; j++)
      filCreateIterFilament(pool[i*n/p], i, j);

  filStartFilaments();
  ...
}
```

Figure 8: An example Jacobi Iteration program using the Filaments package.

4.2.2 Filaments Model and Implementation

Each *filament* is a lightweight, non-blocking, stackless thread that is an independent unit of work. Each one may also contain a shared function pointer and the arguments to that function. For an iterative filament, each phase contains the pointer to the function called by each filament. In addition, a fork-join filament contains a pointer to its parent, a counter representing the number of children, and a place to the store the return value (if any). One server thread per processor executes the filaments. Filaments also uses macros and inlining to reduce the overhead and eliminate loss of compiler optimizations introduced by procedure calls. In addition to this, there are optimizations specific to either the fork-join or the iterative type.

After a number of fork-join filaments are created, any filament created that would exceed a user-specified pruning threshold, `p`, are run sequentially, eliminating any overhead due to the creation of further parallelism. If the number of existing filaments drops below `p`, the next filament created will include the necessary parallel code. If `p` is less than the number of processors in the system, some processors will remain idle. With an algorithm such as the Fibonacci sequence where the work per filament is small enough that a condition statement would comprise a substantial percentage of work per filament, another function header, `filCreateFJFilamentOpt(...)`, can be used. It eliminates the condition on `p` once the threshold is reached by executing an equivalent sequential function provided by the programmer. Since this function lacks any parallel code, the subtree will run to completion on that processor without creating additional parallelism[7]. Filaments accomplishes work stealing of fork-join filaments with a simple round-robin scheme of taking filaments from the head of a processor queue, using locks for synchronization.

The effects of the cache on the efficiency of programs has also been considered by the designers of Filaments. For an iterative program, all filaments are created before the runtime system starts. As a result, for work done by shared code, the code pages should safely remain in the cache. Next, for an iterative phase, the filaments in that phase are executed in a 'back-and-forth' pattern. By executing the last filaments of phase `k` first in phase `k+1` the filament descriptors and argument data for the first few filaments should still be in the cache during the later iteration. Finally, Filaments includes an option to analyze the pattern of execution of iterative filaments. If a regular pattern, such as a loop, is discovered, the Filaments package, during runtime, switches to code that iterates over the filaments to generate arguments in registers rather than loading them from memory.

With the help of the programmer or a compiler, the Filaments scheduler can also overlap computation and communication. If a filament incurs a remote access, instead of continuing to the next filament, the entire pool containing that filament is suspended. Because of data locality, by placing filaments that access the same data into the same pool, repetitive remote accesses on the same page can be circumvented. While the pool is blocked, the Filaments scheduler can execute filaments from an alternate pool (if available) until the remote data is available. Finally, pools that incurred remote accesses on previous phases are scheduled to execute first in subsequent iterations. By doing so, remote communication is incurred sooner and the scheduler can execute other pools during communication.

The Filaments library does not include explicit locks and as a result, *reductions* are used ensure the safety of shared memory accesses. A reduction variable is a logically shared variable for which a copy exists on each node. Access to reductions is restricted to the use of library routines and may only be updated using an associative/commutative operators such as addition or maximum. Performing a check on a shared memory system causes a single node to step through the array of values performing the reduction operation for each value. Filaments uses a standard fan-in algorithm to collect the values on a distributed system and then broadcasts the results to all nodes.

[7] A potential problem with load imbalance may arise with pruning (see section 3.2.4 for details).

4.2.3 Advantages and Disadvantages of Filaments

The differentiation of Filaments between fork-join and iterative filaments is both its greatest advantage as well as its most notable disadvantage. Primarily, this separation allowed the developers of Filaments to optimize each type of filament individually without sacrificing performance of the other. As a result, algorithms using either type of filament run efficiently and produce good speedup. Also, Filaments is implemented with routines for both shared and distributed memory systems. Through this diversity with fork-join and iterative filaments, the package runs efficiently on a wide variety of applications (see section 4 for performance results).

Another advantage of the Filaments package is the ability to designate which node or processor to run each filament. This allows the programmer or a compiler with knowledge of the application to divide work initially to reduce work stealing and/or communication costs. Similarly, the programmer or compiler could divide work among iterative pools to overlap communication and computation (described in section 3.2.2). This is most easily done by a compiler that can analyze memory access patterns. It can be done by hand, although it can be cumbersome to code.

The efficiency of pruning fork-join filaments when using `filCreateFJFilamentOpt()` is a valuable optimization to the Filaments package, but if the call tree is unbalanced, a heavily loaded branch may cause other processors to remain idle while one processor completes a majority of the work. Filaments, upon reaching the pruning threshold, will execute an entire subtree of the call graph sequentially. If there is no additional work available when other processors complete their work, they will remain idle while one processors finishes.

The distinction between filament types, while efficient, forces the programmer to learn the interface for both iterative and fork-join filaments. The Filaments runtime system also restricts the programmer to using one type of filament for each call of `filStartFilaments()`. In an algorithm that may benefit from both types of filaments, one type would be required to run sequentially, which reduces the potential for parallelism.

It is important to note that filaments are not general threads. They were designed to be smaller and faster than traditional threads, but as a result are more limited. They cannot assume independent control (i.e., if one thread forks another, but does not join, the threads will not run concurrently, as would the general thread model). Also, the programmer is left without the use of standard locks or semaphores that are available with traditional threads, which means point to point synchronization is not possible. The use of reductions is more efficient with iterative filaments, but larger critical sections of code introduce a new burden to the programmer[8].

4.3 Lazy Threads

Lazy Threads was created at the University of California at Berkeley in 1995. It was developed as a patch to `gcc-2.6.3` and runs on the CM5 and a cluster of Sparc workstations. Lazy Threads did not support the shared memory platform used in our tests, but the stack management model, *stacklets*, included in the package provides an efficient framework for fine-grain thread creation and execution.

4.3.1 Lazy Threads API

Lazy Threads is integrated with the compiler and provides several compile-time options including different representations of work in the parent, work queuing methods, and thread state management techniques (including *stacklets*). Compilation creates several executables – each of which take advantage of various combinations of the compile-time options.

[8]The programmer could use reductions to implement a basic lock, but this would be inefficient as compared to standard locks in other packages.

```
forkable int fib (int n) {
  int x, y;

  if (n <= 2)
    return 1;
  {
    pcall() x = fib(n-1);
    pcall() y = fib(n-2);
    join();
  }
  return (x+y);
}

int mymain (int argc, char *argv[]){
  ...
  all_start_threads() result = fib(n);
  ...
}
```

Figure 9: An example Fibonacci Sequence implemented using Lazy Threads.

Implementing a parallel algorithm using the Lazy Threads system requires inserting library calls into the program. In the Fibonacci sequence example in Figure 9, the `forkable` keyword indicates that the procedure is potentially parallel, `pcall()` indicates the creation of parallelism, and `join()` synchronizes the running threads. The keywords `pcall` and `join` may only be used within a `forkable` function. Finally, `all_start_threads()` initializes the runtime system and runs the first `forkable` function.

4.3.2 Lazy Threads Model and Implementation

The goal of Lazy Threads is to efficiently support an unrestricted parallel thread model by minimizing the overhead required to create and manage parallelism in a program. In order to do so, the developers of Lazy Threads implemented several different thread and stack management models—most notably the implementation of a stack layout called *stacklets*. A stacklet is a contiguous region of memory on a single processor that acts like a normal stack but contains enough extra information to handle independent threads of control if necessary. Multiple stacklets can be allocated on a single stack to form a tree structure commonly known as a *cactus stack*.

Each stacklet (Figure 10) consists of a frame area that holds any activation frames created by procedure calls or forked threads (described later in this section) and a stub that stores the global information necessary to manage the stacklet tree. Inside the frame area, activation frames are allocated and deallocated using standard call-return semantics, updating the frame pointer and stack pointer appropriately. Following this model, if the compiler can determine that a thread will run to completion without blocking, it is executed exactly like a procedure call. If the compiler determines that the thread will block or if an explicit fork is called, the thread is allocated on a new stacklet that can be run independently of its parent. Finally, if the compiler cannot determine exactly what the thread will do, and it has the potential to block, the thread is executed as a *parallel ready sequential call* or *PRSC*. A PRSC contains enough added information to allow independent control if necessary, but will behave exactly like a sequential call if the thread runs to completion without blocking. Simplicity of memory management is preserved in this model because no two frames within the same stacklet will ever contain free space between them.

A new stacklet may be created when a thread has blocked. In Figure 11, the thread executing `P()` forks

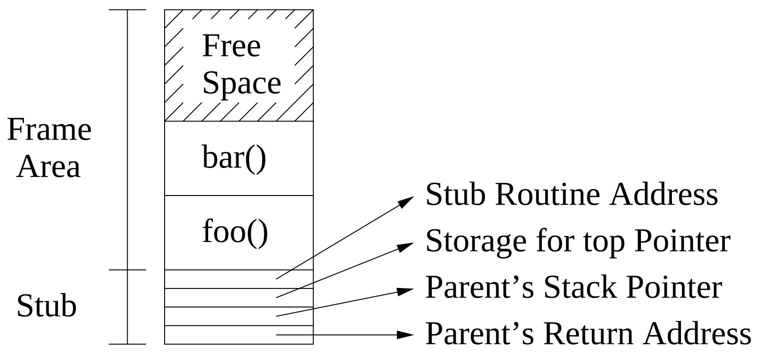


Figure 10: The basic form of a stacklet

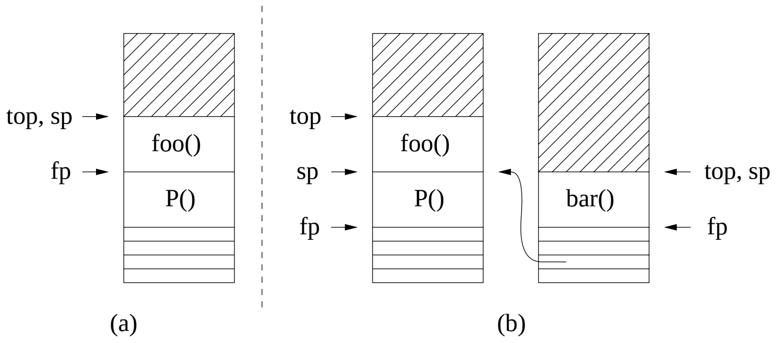


Figure 11: a) The current state of a stacklet where parent `P()` forks `foo()`. b) `foo()` has blocked and control returned to `P()`, which forks `bar()` on a new stacklet.

`foo()` in the first stacklet and `foo()` later blocks. Control returns to the parent `P()`, but an additional pointer *top* remains pointed to the top of the used stack. In order to maintain call-return semantics found in a traditional stack implementation, until `foo()` completes, any future procedure calls or thread invocations, such as `bar()`, are allocated on a new stacklet. Since `P()` is no longer on top of the stack, a simple comparison between the stack pointer (*sp*) and *top* determines if a new stacklet is necessary. To prevent unnecessary overhead, the compiler only adds this check to code where a PRSC is followed by either (1) a procedure call or (2) another PRSC This is because the first PRSC (with the exception of the overflow case described below) should have enough space in the current stacklet to execute like a procedure call. In other words, no other threads (PRSCs) should be blocked above the current frame when the first PRSC is executed.

After `foo()` is resumed and eventually finishes, control must return to the parent to a location different from the original return address. To avoid unnecessary overhead by passing two return addresses, the compiler guarantees that the two return addresses are separated by a fixed number of instructions, allowing a simple address addition to calculate the return point. Within stacklets, both parent and child have access to the child's return address. The parent can then modify the suspension return address if necessary to indicate the proper return location within the parent. Among other optimizations, Lazy Threads uses code duplication to improve the efficiency of the synchronization between the parent and the children in the special case when a join follows a pair of forks. Basically, the parent updates the return address of the child to point to an inlet that reflects the state of the parent. By creating one inlet for each possible state, proper synchronization without additional checks or variables is possible[9].

If enough procedures are called without returning, the activation frames will eventually overflow the

[9]Further details of this implementation can be found in [GSC96].

stacklet. In this case, since there may be more memory available on the stack, a new stacklet is allocated and the frame is created in the new location. Upon completion, the procedure in the new stacklet returns through a stub handler function specified and control passes back to the original stacklet. The address for the stub handler is stored below the bottom frame of a stacklet where the return address for a procedure would normally be. In essence, the stacklet *returns* to the stub handler which deallocates the now empty stacklet and returns to the parent stacklet (whose address is also stored in the stub).

In Lazy Threads, the creation of parallelism does not always create a full thread. Instead, a *nascent* thread can be created and placed on a queue. A nascent thread consists only of a pointer to the parallel code and can be instantiated at any time by an idle worker. This method of work stealing introduces no additional overhead from creating the thread on the idle worker outright. If there are no nascent threads waiting, a thief may steal an entire idle stacklet, which introduces a larger amount of overhead. The benefits of nascent threads can be seen with an example of two forks followed by a join. The first fork will be instantiated and run immediately and the compiler can create a nascent thread for the second fork. The nascent thread can then be stolen by a second processor if necessary.

4.3.3 Advantages and Disadvantages of Lazy Threads

With stacklets, Lazy Threads has introduced a unique method for thread management that supports an unrestricted thread model with a minimal amount of overhead introduced to a sequential (single-node) program. A worst case overhead caused by stacklet overflow is one notable disadvantage. If a series of procedure calls fills a stacklet, the next procedure call would be allocated on a new stacklet. Consider the case where this procedure returns quickly and is called again repeatedly. A new stacklet would be allocated and deallocated again for each call to the procedure, introducing a large amount of overhead to the program. A similar situation occurs when a blocked thread is on top of the stacklet, and the parent makes a number of forks or procedure calls, creating a new stacklet for each call.

In many cases, this situation can be circumvented during compile time. Since Lazy Threads was designed as a compiler modification for `gcc-2.6.3`, call graph analysis can be performed to determine the best possible size for *all* of the stacklets to minimize the number of stacklet allocations. This analysis also aids the simple memory management scheme, ensuring that each stacklet is used efficiently. While integration with the compiler limits portability to a small degree, the unique form of stacklets would be difficult to implement strictly in a library and far less efficient.

Because Lazy Threads did not support shared memory on the platform used in our benchmarks, we were unable to analyze its performance and measure speedup to compare to the other packages. However, on a single processor, the compiler assistance allows fork and join to be as close to a procedure call and return as possible.

4.4 StackThreads/MP

The original idea for StackThreads/MP was developed in 1994 at the University of Tokyo, but the StackThreads/MP library itself was not released to the public until early 1999. The most recent release includes the StackThreads/MP library, source code, documentation, and patches for `gcc-2.7.2.3` and `gcc-2.8.1`[10], which can be downloaded from the StackThreads/MP web site http://www.yl.is.s.u-tokyo.ac.jp/sthreads/ in a 900K .tar.gz file.

StackThreads/MP requires `gcc` (version 2.7.2.3 preferred) and GNU `awk` (version 3.03), and recommended software includes GNU `make` (version 3.77). Currently, StackThreads/MP runs on SPARC and x86 based Solaris, Alpha and x86 based Linux, Digital UNIX, IRIX, Windows NT, and requires approximately 2 megabytes of disk space.

[10]Due to a bug in the later versions of gcc, StackThreads/MP requires the patch to ensure thread safety.

```
void pfib (int n, int *r, st_join_counter *c) {
  if (n < 2) {
    *r = 1;                    /* Write the result */
    st_join_counter_finish(c); /* Announce current thread completion */
  } else {
    int a, b;
    st_join_counter_t cc[1];   /* Create a join counter */
    ST_POLLING();              /* Work stealing and garbage collection */
    st_join_counter_init(cc, 2); /* Initialize join counter to 2 threads */

    ST_THREAD_CREATE(pfib(n-1, &a, cc)); /* Fork fib(n-1, ...) */
    ST_THREAD_CREATE(pfib(n-2, &b, cc)); /* Fork fib(n-2, ...) */
    st_join_counter_wait(cc);  /* Wait until forked threads are finished */

    *r = a + b;                /* Write the result */
    st_join_counter_finish(c); /* Announce current thread completion */
    ST_POLLING();              /* Work stealing and garbage collection */
  }
}

int st_main() {
  int n = 20, answer;

  st_join_counter_t c[1];      /* Create a join counter */
  st_join_counter_init(c, 1);  /* Initialize join counter to 1 thread */

  ST_THREAD_CREATE(pfib(n, &answer, c));/* Create initial thread */

  st_join_counter_wait(c);     /* Wait unitl forked thread has finished */

  printf ("Fibonacci of %d = %d\n", n, answer);
  return 0;
}
```

Figure 12: An example Fibonacci sequence program including the necessary StackThreads/MP library routines.

4.4.1 StackThreads/MP API

The StackThreads/MP library implements several methods that are useful to parallel programmer. As demonstrated by the example (Figure 12), a StackThreads/MP program is not simply a trivial transformation from the original algorithm. In order to indicate the creation of parallelism, the macro `ST_THREAD_CREATE()` is used, and to implement a fork-join protocol, an explicit join counter must be created and initialized by the programmer. `ST_POLLING()` is a macro that causes both the work-stealing algorithm and a simple garbage collector to be run. The garbage collector frees dead frames from the top of the stack.

In addition to the join counter, the StackThreads/MP library also includes primitives for synchronization such as semaphores, condition variables and locks, which may be accessed using methods similar to the join counter type. Other interface routines provided by the library include functions for examining the stack contents and some profiling tools.

4.4.2 StackThreads/MP Model and Implementation

StackThreads/MP does not employ the use of an additional preprocessor but instead relies on the library routines and a postprocessor. The postprocessor is an AWK script that performs the following tasks. First, it analyzes stack frame formats for procedure calls and attaches the analysis information to the assembly file. Second, it adds to the epilogue code for each procedure[11]. Specifically, if the procedure blocks, the epilogue code is run (see below), but the procedure has not terminated; hence, the epilogue code must include this case. Finally, the AWK script collects and updates the stack tables from all of the object files. Because of this, programmers are only required to insert a few library calls in order to use the StackThreads/MP package.

The goal of StackThreads/MP is to provide efficient fine-grain threads by reducing thread overhead to that of a procedure call. In a typical procedure call of an imperative programming language, the context of the caller is saved, and the parameters and the return address are placed above the stack pointer (which points to the top of the current frame). Control is then passed to the called procedure by updating the program counter, moving the stack pointer to point to the new top of the stack, and moving the frame pointer to point to the location of the old stack pointer. Once the procedure has finished, the context of the caller is restored and control returns to the parent at the location specified by the return address.

In StackThreads/MP, the stack pointer always points to the first free frame on top of the stack. This enables newly created procedures and threads to be placed in a free frame on top of the stack. Once allocated, the thread can run as any other procedure would. Upon completion the frame is deallocated and the thread returns to the caller just as any standard procedure call does with the addition of a couple of instructions[12] to the epilogue code of the procedure.

When a thread blocks, it sets a flag, executes its own epilogue code to restore the context of the parent and then returns control to the parent. In previous StackThreads/MP versions, the frame was also copied to the heap, but this was slow, and since frames were not guaranteed to resume in the same stack frame where it was originally created, a problem arose for any architectures with absolute addressing. For example, if a frame `f` blocked and was consequently copied off of the stack, a variable `x` in `f` might be in a new location once `f` is resumed, and an absolute address would be incorrect for `x`. In the current version, frames are left in place on the stack when they block[13].

For a thread that has called a chain of procedures, or forked additional threads, this procedure is a little more complicated. Following the example in Figure 13, consider a parent thread `P` who *forks* a procedure `foo()`, which *calls* the procedure `bar()`. If `bar()` at some later point blocks, then the following events occur. First, a frame for the `block()` procedure is placed on top of the stack. Next, `block()` modifies it's own return address to jump directly to the epilogue code of `bar()` and modifies the return address of `bar()` to enter a special handler when returning. The handler then captures the context of `foo`, runs the epilogue code of `foo()` and returns control to `P`. By executing the epilogue code for each procedure in the call tree, the context for the parent thread is guaranteed to be restored to the state when `P` originally called `foo()`. In a similar manner, resuming a blocked thread occurs by reconstructing the call chain to return to the point where the thread was blocked. This is done by recreating the contexts of the blocked frames that are stored during the suspend call.

By using the core library routines provided in the library, a work-stealing algorithm is provided (called by `ST_POLLING()`) that performs a series of blocks and two resume actions (Figure 14). After StackThreads/MP selects a thread `T` to migrate, all threads above `T` on the stack are suspended (blocked). Next, `T` is suspended and the parent thread of `T` resumes the first thread above `T` on the stack. The thief then resumes

[11] The epilogue code is the code executed at the end of every procedure to restore the context of the caller before returning.

[12] The number of instructions executed depends on whether or not the thread has blocked. A thread which has not blocked executes only two additional instructions on a Sparc.

[13] This introduces a few problems such as stack fragmentation, etc. See section 3.5.3 for details.

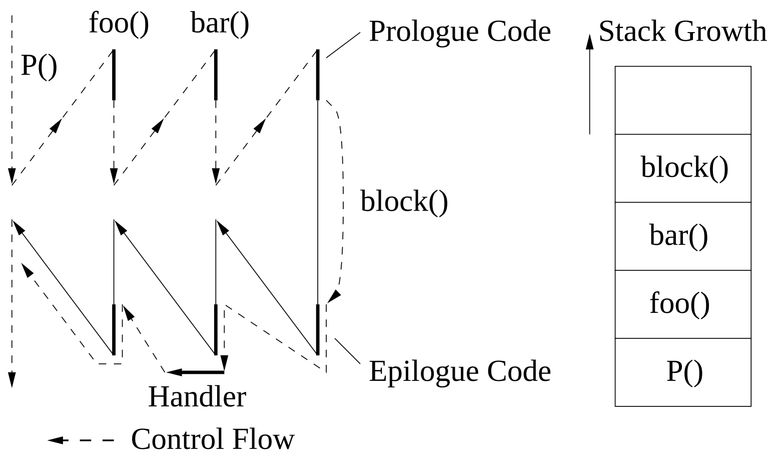


Figure 13: An example of how the chain of epilogue code is executed for a blocked thread. When `foo()` blocks, the block runs the epilogue code for itself, for `foo()`, and for `bar` before returning control to `P`.

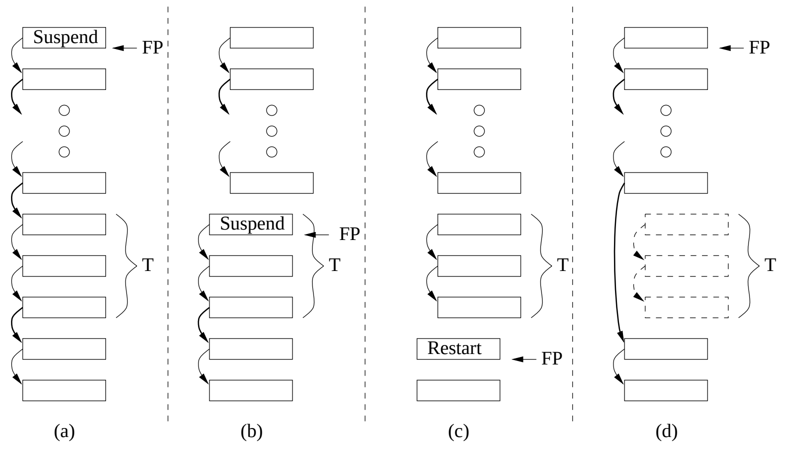


Figure 14: During thread migration, a) any threads above the selected thread `T` (consisting of any number of activation frames) are suspended, b) `T` suspends, c) the parent of `T` restarts the threads above `T`, d) the thief restarts `T`.

`T` using the context saved when `T` suspended. This algorithm, while simple, introduces significant overhead when work is stolen.

In addition to the work-stealing algorithm, `ST_POLLING()` also performs garbage collection. Since blocked threads are left on the stack, subsequent threads are placed above the blocked frames. The lower threads can finish at any time, so the stack may become fragmented. The garbage collector checks the top of the stack for dead frames, deallocates them, and moves the stack pointer down to the highest used frame.

4.4.3 Advantages and Disadvantages of StackThreads/MP

Since each thread is executed at nearly the same speed as a sequential procedure, StackThreads/MP is efficient for programs where threads run to completion without blocking. By only adding a couple of instructions in the epilogue code for each of these types of threads, in many cases, the performance loss for the single-processor version as compared to the sequential program is minimal.

Unfortunately, StackThreads/MP has a few notable drawbacks. First, by leaving blocked frames on the stack, if a thread below the blocked thread finishes, the stack will become fragmented and in the worst case,

```
#include "st.h"

st_mutex_t *A;  /* Global Lock */

void foo() {                    void bar() {
  st_mutex_lock(A);               st_mutex_lock(A);
  ST_THREAD_CREATE(bar());        ST_THREAD_CREATE(foo());
  st_mutex_unlock(A);             st_mutex_unlock(A);
}                               }

st_main() {
  st_mutex_init(A);
  ST_THREAD_CREATE(foo());
}
```

Figure 15: An example that shows how a program with two threads *foo()* and *bar* that runs on a traditional threads package can overflow the stack in StackThreads/MP.

Figure 15 shows an example that demonstrates how the stack can overflow. In this example, thread `foo()` creates `bar()`, and before running to completion, `bar()` recreates `foo()` on top of the stack. This causes an active frame to be on top of the stack at all times. Since dead frames are only deallocated when they are at the top of the stack, this pattern will climb the stack and eventually overflow. On a standard threads package, this example could run indefinitely, but would not crash.

Second, if a thread executes a large number of procedures and eventually blocks far down in the call chain, the epilogue code for each procedure must be executed until control returns to the root node of the thread. Similarly, resuming control of such a thread requires reconstructing the call chain, which introduces a large amount of overhead as well.

Finally, it is up to the programmer to insert `ST_POLLING()` calls to balance the load on the system. While the StackThreads/MP documentation details some theoretical ideas about where to insert polling calls to achieve maximum efficiency, the most efficient placement depends on the structure of the program. Unfortunately, each `ST_POLLING()` call executes about ten additional instructions in addition to any frame deallocation that may occur.

5 Experimentation and Results

5.1 Test Platform

Our test platform was a SUN Enterprise 3000 with two 248 MHz processors and 512 MB RAM. The machine runs Solaris 2.5.1, and we are using `gcc-2.7.2.3`[14], GNU `make` version 3.76.1, and GNU `ld` version 2.9.1. Since each package had a separate set of requirements, this was the most common environment for our tests.

5.2 Micro Overhead

In order to get a better understanding of the benchmark results, we measured three key thread overheads. Figure 16 summarizes these results. The first row in the figure demonstrates that most packages do not introduce any additional overhead for traditional procedure calls. The alternate stack protocol in Lazy

[14]The exception to this is Lazy Threads, which was built into `gcc-2.6.3`

Overhead(ns)	Sequential	Cilk	Filaments	StackThreadsMP	Lazy Threads
Empty Procedure	16.14	16.14	16.14	16.14	57.35
Empty Thread	N/A	2276	135.3	77.63	87.90
Joining Threads	N/A	598.2	335.5	623.7	1 instruction

Figure 16: Comparison of overheads in nanoseconds for each package.

Threads accounts for a majority of the overhead produced, but some loss of performance is due to the version of `gcc` that is used as a base compiler for the package (Lazy Threads uses version 2.6.3 where the other packages are tested using 2.7.2.3).

Creating and running a single thread introduces overhead, which is shown in the second row. Overheads include thread creation, deallocation and any scheduling costs that may be incurred when running a thread. Our test measures the time it takes for each package to create and run one thread by creating one million threads and taking the average. StackThreads/MP accomplishes one of its primary goals by being the cheapest in this category with Lazy Threads falling only a few nano-seconds behind. In both StackThreads/MP and Lazy Threads a forked thread behaves much like a standard procedure call, resulting in minimal overhead in this test. The overhead introduced is caused by the few instructions inserted to manage the frame pointer (StackThreads/MP) or the stacklet structure (Lazy Threads). Competitively, Filaments, using macros and inlining, introduces a slightly larger amount of overhead for each thread. And finally, due to an implicit `sync` at the end of every `cilk` procedure, Cilk performs the worst of the four packages.

The final entry in Figure 16 represents the amount of time for a parent thread to join with a child thread once it has completed. For Cilk programs, this value is significant because every `cilk` procedure upon termination implicitly joins with any child created in the context of the terminating procedure. StackThreads/MP also introduces a large amount of overhead for this benchmark with its `join_counter` manipulations required for a join. Filaments is less expensive than both. Finally, Lazy Threads outperforms all other packages in this case by only introducing a single instruction of overhead. The Lazy Threads compiler and runtime system ensure that the return address of the child will always return to the proper location in the parent, which eliminates the need for a join procedure.

5.3 Benchmarks

Many parallel algorithms can be broken down into either a divide-and-conquer or an iterative implementation. We measured the performance of two benchmarks from each category in an effort to (1) form a comparison of the sequential program to the single-node parallel program (overhead), and (2) evaluate the speedup with the addition of a second processor. Note that we are *not* performing scalability tests. Our focus is primarily on the overhead of fine-grain programs relative to sequential ones; in general, it is harder for a fine-grain threads package to compete on a single processor against a sequential program than it is for it to get speedup[15] There are circumstances with some packages where speedup is hard to achieve; this is what we are trying to detect on the two-processor experiments (see the iterative benchmarks for more details).

Starting with a purely sequential algorithm, a parallel implementation for each package was developed which is as closely modeled after the original sequential implementation as possible. In order to maintain fairness, we have also attempted to make each parallel implementation as similar as possible. The results of each benchmark are discussed below.

[15] Even early fine-grain implementations of functional languages were able to often achieve perfect speedup on recursive programs [Gol88].

```
double quad (double a, double b, double fa, double fb, double area) {
  double left, right, fm, m, aleft, aright;
  /* compute midpoint m */
  /* compute the area under the curve from a to m (aleft) */
  /* compute the area under the curve from m to b (aright) */
  if (close enough)
    return aleft + aright;
  else {
    /* recurse, forking two new threads */
    Fork(quad, left, a, m, fa, fm, aleft);
    Fork(quad, right, m, b, fm, fb, aright);
    Join(); /* Wait for Forked children to complete */
    return left + right;
  }
}
```

Figure 17: Basic adaptive quadrature divide-and-conquer implementation.

5.3.1 Adaptive Quadrature

Adaptive quadrature is a divide-and-conquer algorithm used to approximate the area under a curve generated by a continuous function. The example code below (Figure 17) demonstrates how the area is approximated by dividing an interval in half and calculating the area of each of the smaller intervals using trapezoids. If the sum of the two smaller trapezoids is close enough to the original interval, computation ceases. For finest granularity, a thread was created for each interval approximated.

All packages ran well in this benchmark producing relatively little overhead and good speedup as seen in Figure 18. One minor problem was encountered in Lazy Threads in that there was a limit to the number of parameters that could be passed to a `forkable` procedure. A simple work-around was implemented by passing a structure with all the parameters. Unfortunately, as seen in Figure 18, this introduced a large amount of overhead to the program.

5.3.2 Fibonacci Sequence

The *n*th Fibonacci number can be computed using the algorithm given earlier (e.g., Figure 2). Each number in the sequence is the sum of the two previous numbers and a recursive algorithm, while inefficient, is a common benchmark that exercises fork-join parallelism. For finest granularity, we created a thread at each recursive call.

The results of the tests (Figure 19) show that Filaments actually runs slightly better than its sequential counterpart. This is an anomaly which we cannot explain; we inspected the generated codes, which were basically identical. The Filaments program *should* perform about identically to the sequential program; this is mainly because the pruning threshold (described in section 3.2.3) limits the number of filaments created and most of the algorithm is executed sequentially[16]. StackThreads/MP and Lazy Threads run with minimal overhead, but Cilk runs quite a bit slower. As with a majority of the test cases, the single-processor overhead for Cilk is quite large. By removing the `spawn` and `sync` keyword from the Cilk program, the execution time returns to that of a sequential program, implying that the thread creation and management accounts for all of the overhead. Also, when the number of threads increases (a higher value of `n` is given), both Cilk and StackThreads/MP slow down tremendously; slowdown does not occur with Filaments.

[16] See section 3.1.2 for details.

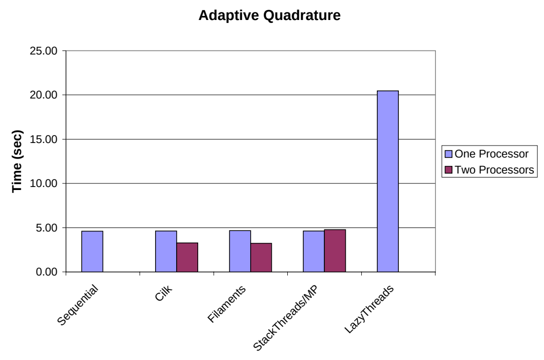
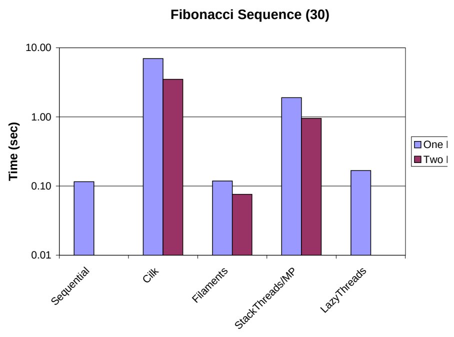


Figure 18: Adaptive quadrature results. Due to a limitation in the number of parameters in a Lazy Threads function, an implementation was used that passed a structure with the necessary parameters. Unfortunately, this introduced a large amount of overhead to the Lazy Threads benchmark. (See text for details.) Note: Lazy Threads does not support shared memory and therefore does not have a two-processor performance result.



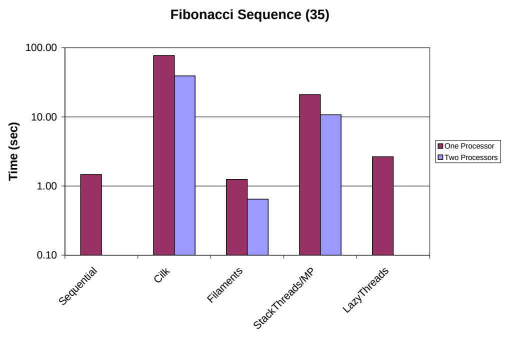


Figure 19: Fibonacci sequence results. Both Cilk and StackThreads/MP introduce a large amount of overhead when the number of threads increase. Filaments pruning threshold limits the number of threads created; it should almost matches sequential performance, but for unexplained reasons is slightly faster. Note that (1) this chart uses a logarithmic scale, so the speedups are near perfect, and (2) Lazy Threads does not support shared memory and therefore does not have a two-processor performance result.

```
void update (int i, int j) {
  B[i][j] = (A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1]) * 0.25;
}

int main (void) {
  Matrix **A, **B;  /* Initialized to a 2-D matrix of size N */
  ...
  for (j = 1; i < N-1; j++) {
    for (i = 1; i < N-1; i++) {
      Fork(update, i, j);
    }
  }
  Join();
  swap(A, B);
  ...
}
```

Figure 20: A generic implementation of the Jacobi iteration which creates one thread per point (`Fork()`)and then waits for all threads to return (`Join()`).

5.3.3 Jacobi Iteration

The Jacobi iteration is a finite difference method used to solve Laplace's and other similar equations. It iteratively updates each point based on its neighbors until convergence. We achieve the finest granularity by creating a thread for each point in the matrix as seen in the example in Figure 20.

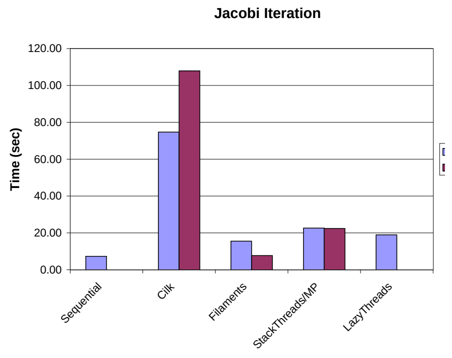
Unfortunately, this naive implementation does not provide speedup in either StackThreads/MP or Cilk (Figure 21). After verifying that roughly half the work was being done by each processor, we determined that because both of these packages rely on polling to achieve parallelism, the simple implementation produces significant overhead from work stealing and that the application actually runs slower than its sequential counterpart. In essence, when work is stolen in a divide-and-conquer algorithm, a thread and all of its children are stolen. On the other hand, in an iterative algorithm such as a Jacobi iteration, all of the threads are on a single level and a work stealing request only takes a single thread. This causes repeated work stealing requests to occur.

In order to efficiently use Cilk and StackThreads/MP it was necessary to rewrite the Jacobi benchmark in a divide-and-conquer style seen in Figure 22. With a more complicated algorithm, this transformation may be more challenging, and because of the added procedure calls and maintenance involved with a recursive programs, the implementation runs slower than its sequential counterpart (Figure 21). This program does, however, produce more efficient speedup for Cilk and StackThreads/MP which can also be seen in the results in Figure 21. However, neither implementation runs as efficiently as the original sequential version — even after a second processor is added.

5.3.4 LU Decomposition

LU decomposition is the process of solving a system of equations by transforming a matrix `A` into two sub-matrices `L` and `U` that are in lower and upper triangular forms respectively. At that point, the equation $Ax = b$ is easily solvable using forward substitution. Our test uses partial pivoting and creates a single thread for each row of the matrix. The algorithm for a standard parallel LU benchmark can be seen in Figure 23.

Much like Jacobi, this iterative style algorithm produces poor results (Figure 24) in both Cilk and Stack-



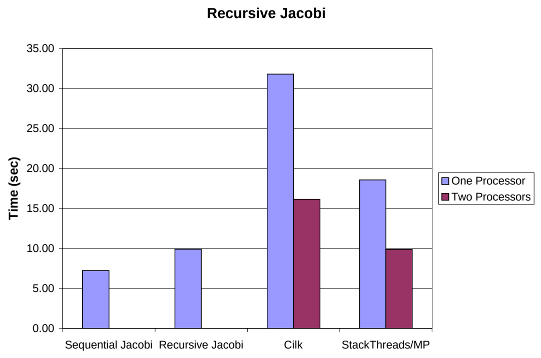


Figure 21: Jacobi iteration results. The first chart shows that Cilk performs poorly with an iterative style program. The second chart demonstrates that while slower, the recursive style of programming produces more efficient speedup for both StackThreads/MP and Cilk. Note: Lazy Threads does not support shared memory and therefore does not have a two-processor performance result.

```
void updateR (int StartI, EndI, StartJ, EndJ) {
  int MidI, MidJ;

  if (EndI == StartI)
    B[i][j] = (A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1]) * 0.25;
  else {
    /* Divide the Matrix into 4 smaller parts and recurse on each part */
    MidI = (EndI - StartI) / 2;
    MidJ = (EndJ - StartJ) / 2;
    Fork(updateR, StartI, MidI, StartJ, MidJ);
    Fork(updateR, StartI, MidI, MidJ+1, EndJ);
    Fork(updateR, MidI+1, EndI, StartJ, MidJ);
    Fork(updateR, MidI+1, EndI, MidJ+1, EndJ);
    Join();
  }
}

int main (void) {
  Matrix **A, **B;  /* Initialized to a 2-D matrix of size N */
  ...
  Fork(updateR, 0, N, 0, N);
  Join();
  swap(A, B);
  ...
}
```

Figure 22: A recursive (divide-and-conquer) implementation of the Jacobi iteration.

```
void parallel_lu(int myrow, int iter, int pivot, double **a) {
  int i = 0;

  if (pivot != iter)
    swap (&a[iter][i], &a[iter][pivot]);
  for (i = iter+1; i < n; i++)
    a[myrow][i] = a[myrow][i] - a[iter][i] * a[myrow][iter];
}

int main (int argc, char** argv[]) {
  ...
  for (k = 0; k < n-1; k++) {
    max = a[k][k];
    pivot = k;

    for (i = k+1; i < n; i++)
      if (fabs(a[k][i]) > fabs(max)) {
        max = a[k][j];
        pivot = i;
      }

    if (pivot != k)
      swap (&a[k][k], &a[k][pivot]);

    for (j = k+1; j < n; j++)
      a[k][j] = a[k][j] / a[k][k];

    for (i = k+1; i < n; i++)
      Fork(parallel_lu(i, k, pivot, a));

    Join();
  }
  ...
}
```

Figure 23: A basic implementation of LU decomposition.

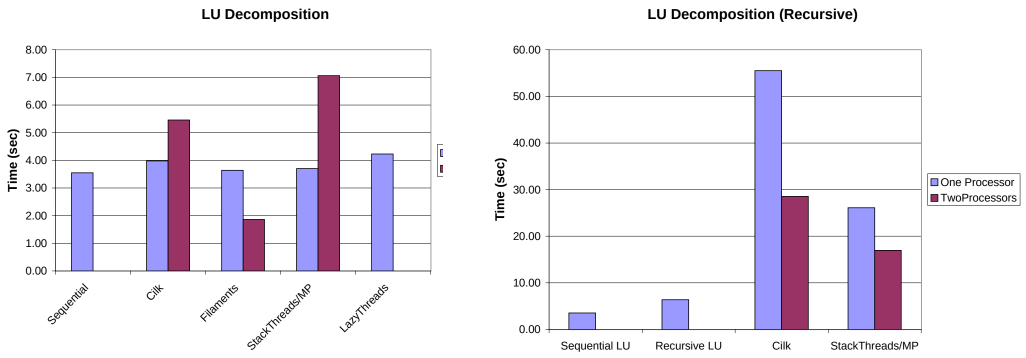


Figure 24: LU decomposition results. Again, Cilk and StackThreads/MP are inefficient with a second processor. With a recursive style program the speedup improves, but the single processor time is much slower. Note: Lazy Threads does not support shared memory and therefore does not have a two-processor performance result.

Threads/MP. Again, a divide-and-conquer algorithm would produce better results. Unfortunately, due to the partial pivoting in this version of LU, a recursive solution is not a straightforward transformation from the original version since the pivoting must be done sequentially. Aside from the pivoting, the LU algorithm in Figure 23 can be transformed in a manner similar to the recursive Jacobi iteration above (Figure 22). As seen by the results in Figure 24, Cilk and StackThreads/MP produce better speedup, but they still do not run as efficiently as the sequential version of LU.

5.4 Discussion

Two significant conclusions can be drawn from the performance numbers of the previous section. First, on our benchmarks, a single processor implementation of Cilk and StackThreads/MP applications introduces a significant amount of overhead to the program. While multithreaded applications always introduce some overhead for thread creation and scheduling, Filaments and Lazy Threads outperform the other two packages. For Cilk, this overhead is caused at least in part by the implicit `sync` call at the end of every `cilk` procedure. StackThreads/MP, on the other hand, creates overhead in the `join_counter` manipulations required in fork-join applications. Very fine-grain applications written using these two packages only amplify this overhead and make them unsuitable unless the granularity is larger. On the other hand, both Cilk and StackThreads/MP perform well with threads with larger workloads. Indeed, Cilk may well be the package of choice for medium-grain recursive parallel programs because of its excellent scalability.

The second conclusion that can be drawn from the performance charts is that while most of the packages produce significant speedup for recursive (divide-and-conquer) style algorithms, only Filaments and Lazy Threads includes the structure and the optimizations for producing efficient *iterative* fine-grain programs. While Lazy Threads does not support shared memory, it is our opinion that the structure of stacklets and the efficiency of the package would produce efficient speedup if shared memory were supported. Both Cilk and StackThreads/MP use a polling method of work-stealing which are (for reasons described in Section 4.3.3) inefficient for iterative programs.

We believe that Lazy Threads has the best potential when considering the combination of an unrestricted threads model as well as an efficiently performing package. However, the cost is a very complicated im-

Threads Package	Thread Model	Package Impl.	1-Processor Performance	Recursive Parallelism	Iterative Parallelism	Load Balancing
Cilk	Restricted	Library[a]	Poor[b]	Efficient	Inefficient	Yes
Filaments	Restricted	Library	Good	Efficient	Efficient	Some[c]
Lazy Threads	Unrestricted	Compiler +Library	Average	N/A	N/A	Yes[d]
StackThreads/MP	Unrestricted	Library[e]	Average	Efficient	Inefficient	Yes

Figure 25: Summary of features for each package

[a]Cilk employs the use of a preprocessing script to convert `cilk` keywords into C-statements.

[b]For fine-grain programs.

[c]Filaments load balances on fork-join filaments, but not iterative filaments.

[d]Although Lazy Threads does not support shared memory, work stealing stacklets or nascent threads are provided (See Section 3.3 for details).

[e]StackThreads/MP includes an `awk` script that performs postprocessing on the generated assembly files.

plementation interwoven with the (`gcc`) compiler, which may be very difficult to port. Although thread creation takes longer than we expected, which causes inferior performance when compared to Filaments, we do not believe this overhead is inherent. Filaments is a good choice for the specific subset of applications it supports, at the expense of a relatively complicated API and restricted thread model. StackThreads/MP is similar to Lazy Threads without the implementation headaches of modifying a large compiler. However, there is no way to avoid additional code in thread creation as well as synchronization, which hurts efficiency on very fine-grain programs. Finally, Cilk is an excellent choice for medium-grain parallel programs.

6 Conclusion

This paper has presented a detailed overview of a number of user-level thread packages that efficiently support fine-grain threads. With the ever-increasing availability (and rapidly decreasing cost) of shared-memory multiprocessing environments and the simple and efficient load balancing provided by larger number of threads, this type of research is becoming more prevalent and more important to application level programmers.

Figure 25 provides a summary of the packages evaluated in this paper and indicates a number of details about each one. Beginning with the traditional thread model, only Lazy Threads and StackThreads/MP fully support the functionality of the traditional thread model. Both Cilk and Filaments restrict the thread model to help achieve efficiency. Neither of these packages allow threads to block and in doing so, they eliminate any overhead created by context switching (except when threads are initially scheduled). Filaments, however, is the only package that does not provide the programmer with a locking mechanism or other form of mutual exclusion synchronization.

With the exception of Lazy Threads, all four packages are implemented *around* the compiler. Cilk uses a preprocessor to convert keywords into C-statements and StackThreads/MP uses a postprocessor to add additional stack management information to the intermediate assembly code. All packages use runtime library calls, but only Lazy Threads is built into the `gcc` compiler giving it a unique advantage over the other packages. Intimate control over stack management and the elimination of extra calls required to access a library can reduce the overhead for multithreading to a minimum.

Finally, each package includes some form of load balancing within the runtime system. Filaments, while providing work stealing for fork-join filaments, does not implement a runtime load balancing mechanism for iterative filaments. To its advantage, however, Filaments does provide the programmer with sufficient means to divide the work among processors initially. Both Cilk and StackThreads/MP fail to do so and instead rely on polling to produce a balanced load, which introduces additional overhead.

Each of the four packages performs well under certain conditions. Cilk is highly scalable and works extremely well with coarser-grain threads. Filaments provides efficiency in iterative applications as well as fork-join programs, at the cost of a more complicated and restricted model. StackThreads/MP reduces the cost of thread creation to that of a procedure call, supports an unrestricted thread model, and is relatively simple use; however, there are significant hidden overheads in very fine-grain programs. Finally, Lazy Threads produces the same procedure call-like efficiency, minimizes overhead for synchronization, and still supports the full thread model; however, it does require significant compiler support. As if often the case, the best package depends on the application and the ability of the programmer.

Acknowledgments

The authors are grateful to Seth Goldstein for his cooperation and support in providing us with the Lazy Threads package for evaluation. We also would like to thank Kenjiro Taura for his prompt support of the StackThreads/MP package.

References

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison/Wesley, 2000.

[Ber98] Daniel J. Berg. *Multithreaded Programming with PThreads*. Sun Microsystems Press, 1998.

[BJK$^+$95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 207–216, July 1995.

[CKP93] Andrew A. Chien, Vijay Karamcheti, and John Plevyak. The Concert system – compiler and runtime support for efficient, fine-grain concurrent object-oriented programs. Technical Report R-93-1815, Dept. of Computer Science, University of Illinois, Urbana-Champaign, June 1993.

[Dij65] E. W. Dijkstra. Solutions of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[FLA94] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *First Symposium on Operating Systems Design and Implementation*, pages 201–212, November 1994.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[Gol88] Benjamin Goldberg. Multiprocessor execution of functional programs. *International Journal of Parallel Programming*, 17(5):425–473, 10 1988.

[GSC96] Seth Copen Goldstein, Klaus Erik Schauser, and David Culler. Lazy Threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[HAA$^+$96] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[HL00] Gregory M.S. Howard and David K. Lowenthal. An integrated compiler/run-time system for global data distribution in distributed shared memory systems. In *Second Workshop on Software Distributed Shared Memory*, May 2000.

[LFA96] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using fine-grain threads and run-time decision making in parallel computing. *Journal of Parallel and Distributed Computing*, 37:41–54, November 1996.

[PKB$^+$91] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multithreaded architecture. In *USENIX Conference Proceedings*, pages 65–80, January 1991.

[PKZA] John Plevyak, Vijay Karamcheti, Xingbin Zhang, and Andrew A.Chien. A hybrid execution model for fine-grained languages on distributed-memory machines. In *Supercomputing '95*.

[TTY96] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Fine-grain multithreading with minimal compiler support – a cost effective approach to implementing efficient multithreading languages. In *Proceedings of Workshop on Object-Based Parallel and Distributed Computation*, pages 59–82, 1996.

[TTY99] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating futures into calling standards. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 60–71, May 1999.