INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Active Threads: an Extensible and Portable Light-Weight Thread System

Boris Weissman

TR-97-036
September 1997

Abstract

This document describes a portable light-weight thread runtime system for uni- and multiprocessors targeted at irregular applications. Unlike most other thread packages, which utilize hard-coded scheduling policies, Active Threads provides a general mechanism for building data structure specific thread schedulers and for composing multiple scheduling policies within a single application. This allows modules developed separately to retain their scheduling policies when used together in a single application. Flexible scheduling policies can exploit the temporal and spatial locality inherent in many applications.

In spite of the added flexibility, the Active Threads API is close to that of more conventional thread packages. Simple synchronization is achieved by standard mutexes, semaphores, and condition variables while more powerful parallel constructs can be easily built from threads, thread bundles (collections of threads with similar properties such as schedulers) and user-defined synchronization objects.

Active Threads can be used directly by application and library writers or as a virtual machine target for compilers for parallel languages. The package is retargeted by porting the Active Threads Portability Interface that includes only eight primitives. Active Threads has been ported to several hardware platforms including SPARC, Intel i386 and higher, DEC Alpha AXP, HPPA and outperformed vendor provided thread packages by as much as orders of magnitude. A typical thread context switch cost is on the order of dozens of instructions and is only an order of magnitude more expensive than a function call. This document presents an involved performance analysis and comparisons with other commercial and research parallel runtimes.

Active Threads are used as a compilation target for Sather, a parallel object-oriented language under development at ICSI. Active Threads are also being used as a base for a distributed extension of C++ that supports thread migration.

1 INTRODUCTION

Threads are gaining widespread use as a vehicle to express parallelism on multiprocessors and to improve the structure of uniprocessor applications. Although threads have only recently entered mainstream computing, the concept of multithreading is hardly new. The ideas behind modern thread packages can be traced back to late 60s and early 70s. As early as 1968, Dijkstra showed how cooperating threads could be coordinated by communication at specific points [1 1]. Programming languages, such as Concurrent Pascal [18] addressed similar issues in mid-70s. However, it was not until the 90s that multithreading started to gain wider acceptance. While in 1991 no commercial operating system provided support for multithreading, in 1997 it is available in some form on most platforms. It is not surprising that the acceptance of the multithreaded model coincides with the emergence of relatively cheap symmetric multiprocessors. Multithreading can exploit the newly available processing power [28].

Many modern operating systems and programming languages provide some support for threads. Emerging platform independent thread standards such as POSIX 1003.4a (recently renumbered POSIX.1c) [23] are supported by multiple vendors. Still, most proprietary thread systems remain too heavy-weight for fine-grain parallel programming.

Active Threads is a portable light-weight thread package for uni- and multiprocessors that promotes compositional software development. It can be used both directly as a thread library or as a virtual machine for compilers for parallel languages. Active Threads were designed to facilitate portable encapsulation of thread scheduling policies while preserving high efficiency and temporal and spatial locality .

This document is organized as following. The remainder of this section discusses the basic multithreading issues and gives a short survey of related work. Section 2 presents Active Threads' main goals. Section 3 introduces the Active Threads programming model and architecture. Section 4 presents a simple example that uses the Active Threads API and discusses relevant scheduling and locality issues. Section 5 discusses implementation and portability issues. Section 6 compares the functionality of Active Threads with that of other parallel runtime systems. Section 7 compares Active Threads performance on a set of benchmarks with that of several commercial and research thread systems. Section 8 provides a detailed performance study of several Active Threads applications and compares their performance with the implementations based on other parallel runtime systems. The full Active Threads API is presented in the Appendix.

1.1 Multithreading Issues

Operating systems are responsible for the creation and scheduling of processes. The allocation of hardware resources to processes is also controlled by the OS. However, multithreading within a process could be performed either by the operating system or by user-level code. There are some trade-offs between kernel-level and user-level thread management.

Benefits of user-level threads:

- *Performance.* Basic thread operations are performed entirely at the user-level and need not incur the kernel trap overhead. The available fast thread primitives make it possible to express fine-grain parallelism naturally pr esent in many application domains. If thread primitives are expensive, applications must be restructured (if possible) to reduce the number of threads and the amount of synchronization. Such restructuring may result in increased data structure complexity because features that are naturally expressible with threads will be implemented in data structures.
- *Flexibility.* User-level threads can be customized on a per-application basis. They need not be overly general, while kernel threads must provide an array of services to satisfy all common uses.
- *Simplicity.* Since user-level threads are easy to extend, the basic thread system need not be complex. For instance, very few threaded applications actually require thread preemption enforced by many kernel-level thread systems. Thread preemption complicates a thread system and has strongly negative performance implications [2].
- *Compositionality.* User-level threads could be pushed one step further and customized on a per-data structure rather than a per-application basis. Data structures and their associated thread policies can be bundled up in modules and accessed concurrently by applications. Kernel threads, on the other hand, must provide a general scheduling policy with the priority scheduling that utilizes a single global priority space being most common across modern operating systems. Since kernel-level threads are inherently non-extensible, the scheduling policy must be general enough to satisfy most needs. Global priorities hinder encapsulation of modules and development of parallel libraries.
- *Portability.* Applications using user-level threads to implement parallelism can be ported by retargeting the underlying thread package. User-level threads can be implemented even for platforms with no operating system support for mutithreading. The semantics of kernel-level threads as well as thread system interfaces usually varies across different operating systems. Applications relying on platform dependent kernel threads are harder to port.

These characteristics of user-level threads make them a natural compilation target for parallel programming languages. Language support makes thread creation, synchronization, and thread pattern reuse easy. Fine-grain parallelism naturally present in many applications can be expressed by powerful programming language constructs. Moreover, fine-grain threads may be necessary to achieve high absolute performance. While cumulative thread creation and synchronization overhead grows with the number of threads, the absolute performance of an application may improve in a more fine-grained implementation. For instance, section 8.1 examines a case for which more threads means higher performance while coarser grain thread solution delivers better speedups over the corresponding serial case, but inferior absolute performance.

Fine parallel granularity is essential for expressing the semantics of many parallel applications in the most natural way, but it places severe performance constraints on the implementation of the runtime thread system. The thread granularity and trade-offs between thread granularity and performance are further discussed in Section 2.1.

Despite all the benefits, user-level threads are not entirely adequate for I/O bound applications. Kernel threads can be a better choice for I/O intensive applications on most modern operating systems. The benefits of kernel-level threads include:

- *I/O and other kernel calls.* Kernel threads usually perform better in a few important cases (such as GUI or file servers) when kernel intervention is necessary [1]. When a thread traps to the kernel or blocks (for instance, because of a page fault), the OS can reschedule another thread. User-level threads blocking inside the OS kernel keep the processor that can otherwise perform other useful work.
- *Uniformity.* Localizing thread operations in the kernel makes thread system interfaces and implementations less volatile. This, in turn, can facilitate creation of "standard" tools such as debuggers, linkers, etc. [2].

It is important to note that the inability of the user-level threads to perform well in the presence of I/O and other OS kernel activity is not inherent in the user-level thread model. It is merely an artifact of the inadequate support of the modern operating systems for user-level multithreading. Poor performance of the user-level threads is a consequence of the lack of feedback from the OS kernel. Several solutions that bridge the gap between the OS kernel activity and user-level thread scheduling were proposed. Marsh, et al. suggested the use of software interrupts and shared OS kernel/user-level thread schedulers data structures to communicate the kernel events to user-level thread schedulers [34]. Anderson et al. offered a *scheduler activation* mechanism as a better abstraction than kernel threads to support user-level management of parallelism [1]. Research prototypes for both solutions were built [1][34] and showed that efficient user-level multithreading is possible in the presence of I/O activity. However, mainstream OS currently provide no support for cooperation between user-level threads and the OS kernel.

The performance of kernel threads, on the other hand, is inherently worse than that of user-level threads. The main reason is the kernel trap overhead for all thread operations. This aspect of kernel threads is explored in depth in [1].

While no mainstream commercial operating system currently fully supports the kernel feedback mechanisms, Sun Microsystems announced that Solaris 2.6 (to be released in late 1997) will include scheduler activations [55].

1.2 Prior Work

Many modern multiprocessing operating systems such as Solaris [42][47], Mach [12] and Windows NT [10] support kernel threads. Since the performance of kernel-level threads is usually at least an order of magnitude worse than that of user-level threads, user threads have been implemented on top of kernel threads in many systems: CThreads for Mach [12], WorkCrews for Topaz [63], Solaris Threads for Solaris OS [47].

Different aspects of user-level thread systems have been explored in the literature. PRESTO [3], a system from the University of Washington, emphasized the value of an object-oriented design for a thread system. PRESTO came with a preemptive scheduler, but the

user could replace it with an application-specific scheduler . The same applied to synchronization objects. Later versions of PRESTO tried to exploit locality by preferentially scheduling threads that previously ran on a particular CPU over threads that executed elsewhere, although no attempt was made to use the semantics of thread operations for locality based scheduling.

Although the Berkeley Threaded Abstract Machine (TAM) [9][46][13] did not provide a general-purpose thread system, it explored some locality aspects of thread-scheduling on distributed architectures. TAM was designed as a compiler substrate for lenient (non-strict) programming languages. In the TAM model, a program is a collection of *codeblocks*, where each codeblock consists of possibly several nonblocking threads. Threads are restricted: a thread is a sequence of instructions with no jumps or synchronization points; synchronization occurs only at the top of a thread All threads in a codeblock run to completion and the last thread to run, schedules the next codeblock to execute. This scheduling policy enhances locality by concentrating on a single frame that binds logically related threads together as long as possible. A TAM based implementation of Id90 (a strongly-typed functional language), demonstrated performance about an order of magnitude higher than LISP and an order of magnitude lower than C on a single processor [9].

Lazy Threads [14][15], a follow-up project to TAM, concentrated on reducing the overhead of a parallel call while preserving the generality of threads. Similar to TAM, Lazy Threads is a compiler substrate - it relies heavily on compiler optimizations and cannot be used directly as a parallel library. However, unlike TAM, Lazy Threads supports blocking threads. The central idea is to specialize the representation of a parallel call so that it can be executed as a parallel-ready sequential call. In cases when parallelism is excessive and threads do not block this results in parallel call overheads close to those of sequential calls. Blocking of a child thread leads to the creation of a new thread for a parent. The system relies on the compiler support for non-standard call/return sequences. The default sequential execution of threads is motivated by the experience with the non-strict functional language Id90 which showed that the majority of potentially parallel calls can run sequentially. The system was implemented on the CM5 and a single processor SPARCstation.

StackThreads [56][57] attempts to reduce the thread creation overhead by allocating the activation frame for a new thread first on the stack and moving it to the heap if the thread blocks. Similar to Lazy Threads, non-blocking threads incur little overhead. All calls, sequential and parallel, use the same representation. This somewhat increases the direct function call/return overhead for the general case. The system was implemented on a single processor and required compiler support.

In a similar effort to drastically reduce thread creation overhead, many other researchers have turned to simpler parallel models. Leapfrogging [64] restricts the behavior of the program in an attempt to reduce the cost of futures in functional languages. The technique assumes that futures do not create other futures (which can be thought of as child threads), unless they directly depend on their values. This guaranteed dependency allows the runtime system to avoid creation of new stacks for all threads. A single worker thread can be

used to evaluate several futures in the presence of a linear dependency. The semantic restriction that enables leapfrogging reduces the usefulness of this technique for explicitly parallel object-oriented languages which generally support very expressive threads and synchronization constructs.

In an another attempt to reduce the thread creation overhead, many runtime systems restricted the semantics of threads even further by supporting only non-blocking threadlets that, once started, run to completion (Filaments [30], Cilk [5][6], Multipol [66]). Although non-blocking threadlets do provide low thread creation overhead and generally do not require memory for thread stacks, they are not adequate in their expressiveness to be easily adapted as a compilation target for most modern explicitly parallel object-oriented languages (including Java, Ada, CC++, and the locally designed language Sather). Using non-blocking threads requires shifting to a radically different programming style, such as continuation passing or implicitly parallel functional programming.

Other relevant systems providing parallel programming with threads include Ariadne [35][36] and Mthreads [48]. Ariadne is a general-purpose portable thread system from Purdue University that achieves portability by building on top of C library setjmp/longjmp primitives. It is similar in functionality to proprietary systems such as Solaris Threads, but provides portability across different hardware platforms. Detailed performance comparisons between Ariadne and Active threads are offered in section 7. Ariadne makes no attempt to use the memory hierarchy for thread scheduling.

Mthreads is a user-level thread library for Convex SPP developed at the University of Erlangen-Nurnberg, Germany. The project explored the space of affinity-based scheduling for the Convex SPP 1000 machine (NUMA architecture). It was shown that using runtime cache miss rate information for thread scheduling can have a positive performance impact. All examined scheduling policies used runtime cache misses rather than user annotations for thread scheduling.

QuickThreads [27] provided a set of techniques and tools for building portable user-level thread packages. It defined a low-level layer for portable thread systems. The design and implementation of the Active Threads portability interface was influenced by the ideas and implementation issues explored in [27].

Active Threads has many common features with the thread systems listed above. For instance, applications using other thread packages can usually be mechanically converted to Active Threads. However, there are also important differences. Most of these differences stem from the two distinct goals of Active Threads: to be an efficient portable compilation target for parallel-object oriented languages and to provide good abstractions for building compositional parallel libraries.

2 ACTIVE THREADS: MOTIVATION AND GOALS

The Active Threads system was designed to facilitate high-performance fine-grained platform-independent parallel programming; to enable applications to take advantage of the memory hierarchy of modern machines; to make possible modular and compositional development and performance profiling of thr eaded software components.

2.1 Fine-Grain Parallel Programming

Many existing parallel languages support fine-grain parallelism. These languages encourage the user to express all the parallelism naturally present in the problem. The degree of parallelism could be quite high, dynamic, and independent of the actual number of processors. On the data-parallel side, languages like NESL [4] and HPF [21] provide constructs to express fine-grain nested data parallelism. Contr ol-parallel languages such as CC++ [7], Cilk [5], and Sather [16][49] provide powerful control mechanisms for expressing fine-grained task parallelism.

There are several reasons for the popularity of languages that encourage the fine-grained parallel programming style:

- *Expressiveness.* A natural parallel decomposition of a program maps directly onto the threaded implementation. The implementation reflects the problem's logical concurrency rather than a particular hardware architecture. This generally improves the application structure.
- *Portability.* Fine-grained parallel programs are not very sensitive to the number of available processors. A fine-grained thr eaded application can even dynamically adapt to the changing number of processors. The details of such adaptation can be hidden in the implementation of the thread runtime system and need not be exposed to the programmer.
- *Transparent Load balancing*. The greater the total number of available parallel tasks the higher the probability that at any given time all available processors are busy. The task of load balancing is essentially off-loaded onto to the thread runtime system. The runtime may migrate threads between processors to avoid idle time. Coarse-grained applications, on the other hand, must perform explicit load balancing on the level of data structures.
- *Encapsulation of parallelism*. If the underlying system can effectively support large number (even millions) of threads, parallel patterns can be easily encapsulated in reusable modules with low performance overhead.
- *Communications*. Distributed systems with user-level communications such as Sather can use fine-grained thr eads to mask communications latency. A thread initiates communication and blocks. Another thread is scheduled to run in its place. The initial thread is rescheduled when the necessary remote data is obtained. A similar technique can be used for threads that perform I/O, but this requires a feedback from the OS kernel.

In practice, the degree of parallelism that can be effectively used by applications is limited by the thread system overhead. The performance penalties of fine-grained multithreading come from several sources:

- Thread creation overhead.
- Thread synchronization overhead.
- Cache reload overhead.
- Extra memory usage.

The effectiveness of the thread operations such as the thread creation and context switch effectively determines the granularity of parallel applications. The caching effects are somewhat more subtle and are examined more closely in the following section.

2.2 Locality Effects

An executing thread must get its working set into a processor's cache. The shorter time the threads execute and the more threads block and get rescheduled, the greater the relative cost of such cache reload. For coarse-grain, long-running threads, the cost of building up the cache state is amortized over the entire thread's lifetime. Short lived threads or threads that do not execute long enough before relinquishing the processor suffer from caching effects to a greater degree. Furthermore, the actual caching behavior is highly dynamic and application dependent. The hope for finding a single thread scheduling strategy that will yield good results for all applications is hardly justified. Moreover, applications that are not structured around a single computational kernel are likely to exhibit different behavior with respect to cache at different execution stages.

Finding good thread scheduling strategies is hard because several seemingly conflicting goals must be achieved simultaneously. For instance, to achieve load balance, all threads in the applications can be placed in a central thread repository organized as a FIFO work queue. This guarantees that no processor is idle as long as the central work queue in non-empty. However, this solution is not scalable (due to contention for the single central queue) and usually exhibits poor behavior with respect to cache reuse. If cache reuse is chosen as a main scheduling objective, some processors may be left idle intentionally in the hope to reduce the total number of cache misses. Markatos et. al coined a term *memory-conscious scheduling (MCS)* for such scheduling policies that try to reduce the overhead of loading data into local memory or cache [32].

The research literature exhibits somewhat contradictory evidence of the relative importance of load balancing vs. memory-conscious scheduling. Early work on thread scheduling focused almost exclusively on the goal of load balancing. For example, in the process control policy by Tucker and Gupta [60] and the early versions of PRESTO [3], a single central FIFO queue was used for thread scheduling. A more recent work by Thekkath and Eggers [58] also stated that load balancing was a critical performance factor while memory-conscious scheduling had no positive effect on execution time of fourteen coarse- and medium-grain parallel applications used in the simulation. On the other hand, the work by

Markatos and LeBlanc [32] observed significant improvements (40-60%) due to MCS on a real shared memory platform for several fairly fine-grained parallel applications and kernels. In both studies, each application was structured around a single parallel task such as constructing and traversing a tree.

We believe that the overall application performance is influenced by the interplay of thread granularity, load balancing, and memory locality effects. While for coarser thread granularity load balancing may be a determining factor, as thread granularity increases, locality issues gain greater importance. Recent architecture trends suggest that processors are getting faster at higher rate than memories, and hence the crossover point at which locality effects become predominant will be observed for coarser thread granularity on future architectures than currently.

Active Threads provides a general scheduling mechanism that enables rapid prototyping and implementation of different load balancing and memory-conscious scheduling policies. Different policies can be composed in a single application. Various data structures and parallel control patterns may require different scheduling policies. Such policies can be implemented, profiled, and distributed together with the software modules.

2.3 Parallel Module Composition

Despite overall advancements in creating efficient programming methodologies, high-performance parallel applications are usually developed from scratch and tuned to particular computer architectures. Unlike the serial case, object-oriented techniques alone cannot achieve software encapsulation, code reuse, and compositionality of high-performance multithreaded parallel code. Load balancing, synchronization and locality effects get in the away of parallel module composition. Explicit coding for load balancing and locality violates software encapsulation and hinders software maintainability, extensibility, and portability. However, because parallel programming is inherently harder than serial, parallel pattern encapsulation and reuse is especially desirable from the methodological as well as the practical prospective.

Active Threads offers fine-grained multithreading with system support for modular software development as an efficient platform-independent general-purpose parallel programming paradigm. The system support is necessary to relieve the programmer from the bulk of the work associated with locality maintenance, load balancing, portability and performance predictability across different hardware platforms. System services are also necessary to enable separate development and profiling of parallel modules with predictable effects of module composition. Since it is generally impossible to foresee all software module needs, we envision a system providing such services based upon a general mechanism that can support multiple implementation policies.

The Active Threads system intends to help the user reconcile often conflicting goals of having a clean modular design while preserving portable performance at acceptable levels. This is achieved by providing fast, platform-independent thread primitives and a general scheduling mechanism capable of supporting multiple extensible (and possibly

concurrent) scheduling policies. The granularity of threads and scheduling can be dictated by the application semantics rather than the existing implementation of threads and schedulers. Fast thread primitives including thread creation and synchronization provide for a fine-grained multithr eading programming style. For instance, in Active Threads, thread creation and context switch time are only about an order of magnitude more expensive than a null function call on many hardware platforms. A general thread scheduling mechanism enables rapid prototyping with the following performance tuning of scheduling policies for different data structures and application stages.

It has been suggested earlier that, in theory, the compiler may elect different thread scheduling policies in different portions of the program [9] and different library modules may benefit from custom thread scheduling [27]. However, we are not aware of any practical and portable fine-grain general-purpose thread system that makes this possible on modern symmetric multiprocessors while preserving the inherent temporal and spatial locality. The Active Threads package was designed to fulfill these needs.

2.4 Design Goals

The Active Threads design fulfills the following goals:

1. **High Performance Fine-Grain Multithreading**

   Efficiency of thread primitives. Fine-grain parallel programming imposes restrictions on the runtime system overhead. Section 7 shows that it is possible to perform a thread context switch in just a few dozen instructions on a wide range of modern architectures. Rescheduling overhead must be equally low to keep threads light weight. Similar constraints apply to thread creation and synchronization operations.

   *Memory-conscious scheduling.*Vast improvements in the microprocessor performance of the last decade were not matched by a corresponding speedup in memory and interconnection network latencies. Achieving peak performance on the modern architectures is possible only when the memory hierarchy and placement of threads and data are taken into account. The Active Threads scheduling mechanism was designed to support co-locating threads with their data. Different scheduling policies can be used for experiments with different co-location strategies. Active Threads supports the implementation of static, dynamic, and feedback based placement policies.

2. **Enabling Compositional Scheduling**

   Separating scheduling mechanism and policies. Unlike most thread systems, which provide either a single scheduling policy or a fixed set of scheduling policies, Active Threads provide a general mechanism that allows the coexistence of any number different scheduling policies. Different policies can be developed on a per data structure or module basis. Active Threads captures the dynamic structure of the evolving parallel computation and allows one to schedule threads in a way that reflects this structure. This could be of particular benefit to frequently occurring parallel patterns: nested parallel loops that span all or subsets of processors, collections of cooperating threads working towards achieving a common goal, etc. Active Threads allows us to reason about and specify the behavior of such patterns locally and implement these patterns using the most suitable data structures and

scheduling policies. Any implementation of scheduling policies that provides a set of event handlers for the events generated by the scheduling mechanism can be used in any Active Threads application. This mechanism enables modular development, modular performance profiling, and extensibility of parallel applications.

The majority of existing commercial thread systems either employ a single global priority-based thread scheduling policy or allow a fixed set of thread scheduling classes as in the POSIX standard [23]. Several research systems such as PRESTO [3] and Synthesis [37] targeted the adaptation of scheduling policies for applications needs. PRESTO, a user-level thread system, allowed the user to replace the default round-robin scheduler with a more suitable one. Synthesis provided a round-robin scheduler for kernel threads that could adaptively adjust the CPU quantum allocated for different threads based upon the thread's I/O rate. However, the compositionality aspect of thread scheduling remains largely unexplored.

3. **Programmability**

Simplicity. Active Threads are a general purpose C library that can be used directly or as a compilation target for parallel languages. In spite of many innovative features, Active Threads interfaces were intentionally kept similar to more conventional thread packages. Synchronization is achieved by way of familiar synchronization objects such as spinlocks, mutexes, semaphores, and condition variables. Synchronization objects with unusual semantics can be added to the system.

Portability. Portability was among the initial goals for Active Threads. Active Threads were built to fulfill the need for a portable light-weight threaded compilation target for Sather, a parallel object-oriented language under development at ICSI. Active Threads are ported to new platforms by porting the Active Threads Portability interface which includes only eight primitives (section 5.6) - usually under a hundred lines of C and assembly. Active Threads currently runs on a number of uni- and multiprocessors including SPARC, Intel i386 and higher, DEC Alpha AXP and HPPA. Several more porting efforts are in progress.

3 ACTIVE THREADS: THE MODEL

3.1 Overall Architecture

The major Active Threads components are shown in Figure 1. Active Threads provides the user with the following abstractions:

- threads
- thread bundles
- virtual processors
- synchronization objects

Threads are units of (potentially parallel) execution that share an address space and other system resources. Groups of logically related threads are organized into *thread bundles*, or simply *bundles*. Threads in the same bundle share a common thread *scheduler*. For instance, a parallel loop statement, common in many parallel languages, can be implemented as a bundle of threads. At any time a bundle is associated with a single scheduler.

Active Threads hide hardware dependent variables such as the number of CPUs from the user. Instead, the user is provided with a *virtual processor* abstraction. At the application level, threads can be scheduled to run on virtual processors. The number of available virtual processors is limited only by the word size of the underlying architecture. Users are encouraged to schedule threads that are likely to use the same data to run on the same virtual processor. Such data dependent scheduling annotations are likely to produce substantial performance benefits (section 7). This functionality can also be used to implement high-level programming language abstractions such as *zones* [51] intended to capture locality patterns in a portable manner using user annotations. No precise knowledge of the memory hierarchy is necessary for such scheduling decisions, although some information about the underlying hardware may lead to even greater speedups.

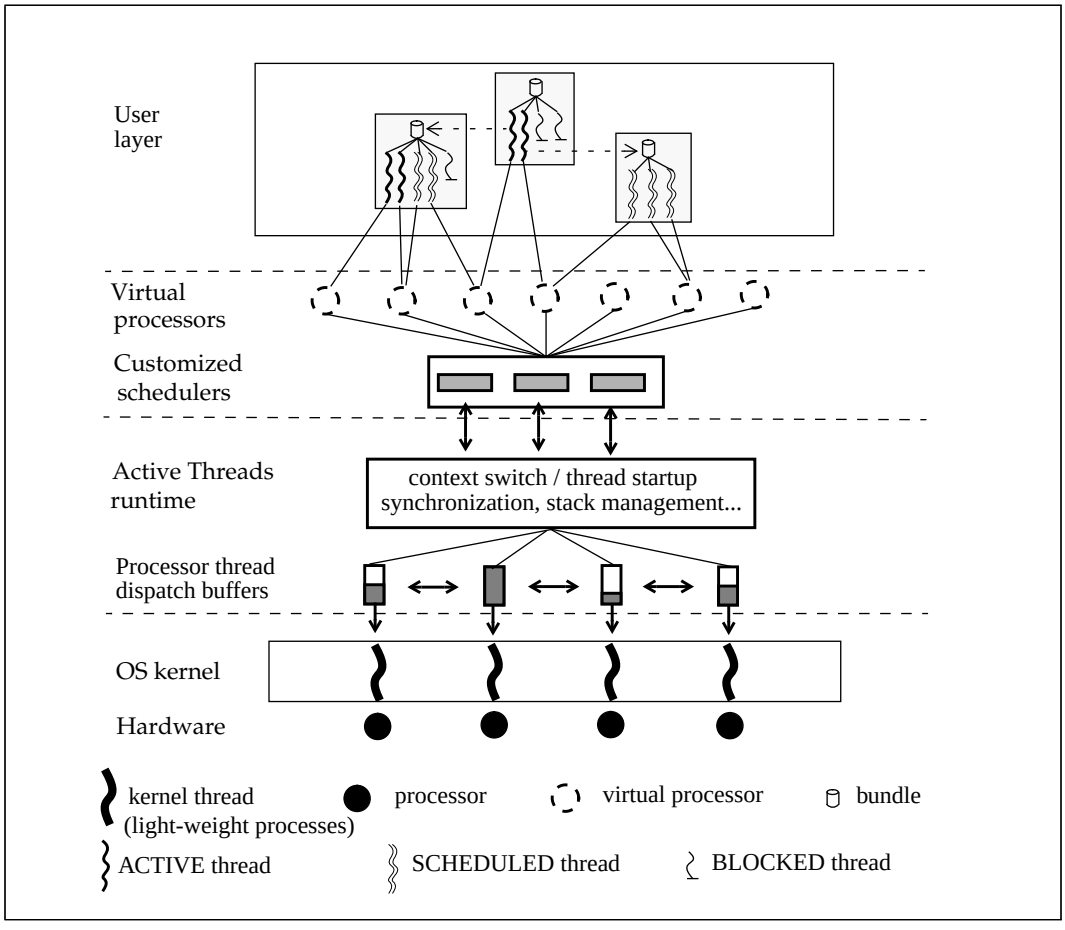

Figure 1: Active Threads Architecture

Virtual processors are multiplexed over available physical processors. All mapping and load balancing details are hidden from the application programmer. However, if needed, the application can influence these decisions by supplying customized schedulers. Active Threads provides a library of various schedulers that can be freely used or extended by the application. The same application can benefit from different thread scheduling policies in different parts or even change a scheduling policy dynamically.

The Active Threads runtime provides basic thread services: thread initialization, start-up and context switch, thread stack management, synchronization primitives such as different locks and semaphores. The runtime is also responsible for keeping processor thread dispatch buffers non-empty to avoid processor idle time.

The Active Threads runtime also deals with various OS specific issues such as the creation and management of kernel threads (*lwps* in Sun terminology [47]or *tasks* in NT [10]) or other entities supported by the OS kernel (for example, *scheduler activations* [1]). Active Threads can also run on top of proprietary user-space thread packages. Much of the runtime is machine independent. Hardware dependent services are captured in the Active Threads Machine-Dependent Layer. Active Threads are ported to new platforms by retargeting the Portability Interface of the Machine-Dependent Layer. Current ports include the following uni- and multi- processors: SPARC, Intel i386 and higher, DEC Alpha AXP and HPPA.

The details necessary for programming with Active Threads are given in sections 3.2 through 3.3.

3.2 Threads

Active Threads are lightweight, non-preemptive, user-level threads with conventional thread semantics: threads are units of execution that share a process address space. Each thread maintains a set of registers including a program counter and a stack that stores the thread's local variables. Threads share all other process resources. There is no enforced protection between threads of the same process.

A thread can be in one of the following states: INITIATED, RUNNABLE, SCHEDULED, ACTIVE, BLOCKED, and DEAD. The state transitions and operations that cause these transitions are shown in Figure 2.

A thread is created by the Active Thread runtime in an INITIATED state. It is then passed to its bundle for scheduling. A thread in the INITIATED state has all information necessary to start execution (such as an entry point and arguments), however it may need to acquire some additional resources for thread startup. For instance, threads in the INITIATED state may be stackless if the scheduling policy uses lazy stack allocation on thread startup rather than thread creation. Such threads may also have thread private storage unallocated.

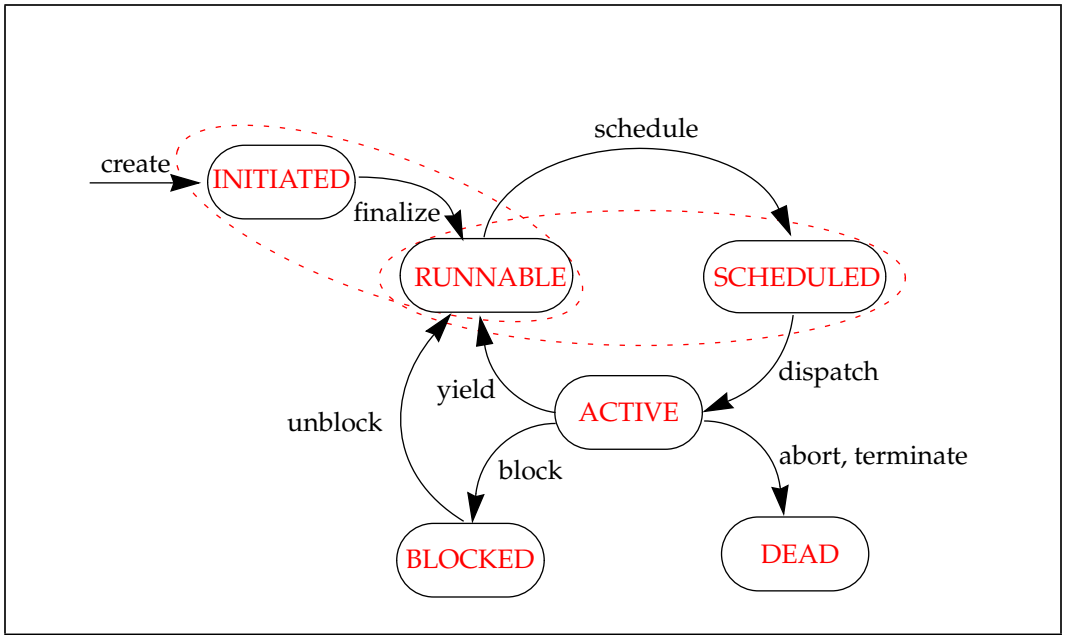

Figure 2: Active Threads State Transitions

Threads in the RUNNABLE state are entirely managed by their bundles and can be freely scheduled for execution by the associated bundles. In general, a user-supplied scheduling algorithm can select a subset of threads in the RUNNABLE state and schedule them for execution causing the transition from the RUNNABLE to SCHEDULED state.

Threads in the SCHEDULED state are no longer accessible to the bundles. They act as a work pool for (physical) processors and are kept in special dispatch buffers. Note that there may be many more SCHEDULED threads than processors. When a processor becomes idle, it dispatches one of the SCHEDULED threads for execution. An executing thread is said to be in the ACTIVE state.

An executing (ACTIVE) thread can yield execution to another thread, block on a synchronization object or terminate. BLOCKED threads are sleeping on the synchronization objects such as mutexes, semaphores, or condition variables until the respective synchronization objects are sent special unblocking signals. After unblocking, a thread becomes RUNNABLE and is passed to its bundle for scheduling. Active Threads also supports user-defined events that, when triggered by the thread in the ACTIVE state, cause a transition to the RUNNABLE state; however the thread is not passed to its bundle. Instead, it is passed to a user-defined event handler. This functionality allows, for instance, to write a thread to a persistent store. It is also used to facilitate migration of threads between SMPs in a distributed system [65].

Threads become DEAD after they terminate or explicitly execute *at_exit*. Table 1 summarizes the Active Threads states and transitions.

State	Semantics
INITIATED	created threads, but possibly missing some resources (i.e. stack or local store)
RUNNABLE	managed by different schedulers; can be scheduled for execution
SCHEDULED	threads in cpu dispatch queues, no longer accessible to schedulers
ACTIVE	executing threads
BLOCKED	threads sleeping on synchronization objects
DEAD	threads after termination, or an explicit call to at_exit()

Table 1: Active Threads states.

Thread state transitions in Active Threads are not radically different from those of other user-level threads packages, such as Solaris Threads [47]. However, the scheduling event mechanism that implements state transitions is a unique feature of Active Threads. So is binding groups of logically related threads together and associating a potentially user-defined scheduler with each thread group.

3.3 Thread Bundles

Thread concurrency and synchronization patterns in many parallel applications are not random. The object-oriented paradigm captures and reuses such recurring patterns especially well. Our experience with Sather, a parallel thread-based object-oriented language [16][49][50], as well as other empirical evidence for non object-oriented parallel languages [9] show that both substantial temporal and spatial locality exists among collections of logically related threads. Exposing thread scheduling decisions to the compiler or a parallel library allows us to exploit common locality patterns in order to minimize negative effects at all levels of the memory hierarchy. Bundles and processor affinity annotations for threads take advantage of such temporal and spatial locality.

We define a *thread bundle* as a collection of semantically related threads with common properties. All threads in a bundle share the same scheduling policy. Scheduling policies for different bundles can be completely independent from each other. A single application may create thread bundles with different scheduling policies such as FIFO, LIFO, priority, processor affinity scheduling, etc. Moreover, a scheduling policy for a bundle is not fixed and can potentially change dynamically.

A thread bundle is created to perform some parallel operation. There may be arbitrary dependencies among the threads in a bundle or among threads in different bundles. However, since a bundle expresses a parallel operation, threads in the same bundle are more likely to depend on each other than on threads in other bundles.

Threads in a bundle can be in different states: INITIATED, RUNNABLE, ACTIVE, BLOCKED, and SCHEDULED. Figure 3 shows a snapshot of a bundle with threads in various states.

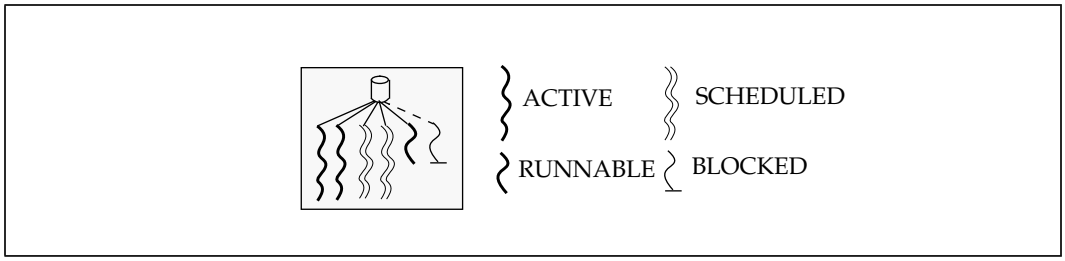

Figure 3: Bundle with threads in different states

When a new thread is created, its bundle is always specified. Thus, at any given time, all threads in the system belong to some bundles. A bundle plays a role of a handle for a collection of threads and performs scheduling for this collection.

Any running thread can create a new thread bundle at any time. In Figure 4, running threads create new bundles (left), which results in a dynamic bundle hierarchy, called a

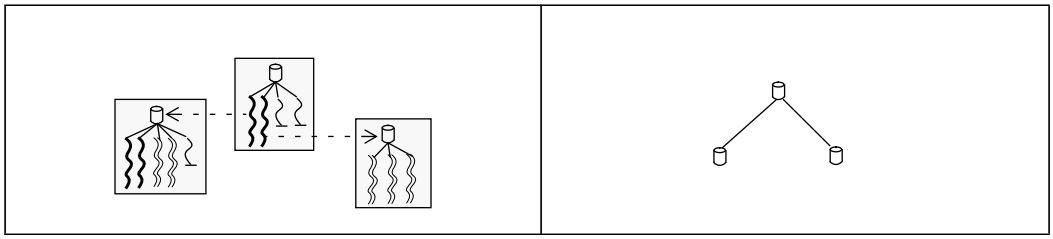

Figure 4: Creation of new bundles (left) and the resulting bundle activation tree (right)

bundle activation tree (right).

Thread bundles form a runtime (dynamic) hierarchy. A thread creating a new thread or a bundle does not need to block. Therefore, an Active Threads application unfolds like a tree, not as a stack. The bundle activation tree diagram of Figure 4 displays only bundles and each node in the diagram could be thought of as a logical unit of a parallel computation.

Bundles are a simple mechanism to communicate the logical structure of the evolving computation to the Active Threads runtime. At some point, threads in the bundle must be mapped onto the physical processors. Per bundle schedulers deal with such mapping and the associated scheduling decisions.

Grouping related threads into bundles leads to two kinds of scheduling decisions:

- scheduling of threads in a bundles
- scheduling of different bundles in a bundle activation tree

3.4 Scheduling

To achieve high performance, Active Threads are designed to be user-level and non-preemptive. Negative performance implications of thread preemption are discussed in [2] and [27]. User-level thread operations provide for greater flexibility and higher performance than kernel-level thread management [2].

There are several main reasons for performance degradation from multithreading:

- The overhead of a context switch, which generally involves saving and reloading a portion of a register file.
- Rescheduling overhead due to consulting the scheduler data structures and identifying the next thread to run.
- Scheduling implications in the presence of dependencies between threads. For instance, it is common for parallel applications to use busy-waiting on some condition. If the thread that signals the condition is not scheduled immediately, other threads that depend on it end up busy-waiting and tying up processors.
- Memory locality effects. If assigning threads to processors is done with no regard for cache contents, problems of several sorts arise. An unblocked thread can be scheduled to run on a processor that does not cache that thread's data. Even if the thread is rescheduled to run on the same processor as before blocking, intervening threads may have corrupted some of the cached state. Alternatively, logically related threads using the same data may execute concurrently on different processors causing large bus traffic due to cache invalidation and false sharing.
- In multiprogramming systems, especially with high loads, the last two problems are aggravated by the OS intervention to multiplex different processes over a fixed number of processors.

Active Threads addresses most of these problems. The context switch overhead is on the order of a few dozen instructions for all supported platforms. A flexible scheduling event mechanism allows us to build and tune a variety of schedulers for particular data structures. This lowers the rescheduling overhead and enables thread scheduling that respects thread dependencies. The busy-waiting problem is addressed by blocking and two-phase synchronization objects supplied by Active Threads. Two-phase synchronization involves spinning on the lock for some duration before blocking. Furthermore, the Active Threads events and runtime provide services necessary to implement a variety of schedulers that facilitate cache reuse.

Unlike most thread management systems, there is no fixed scheduling policy for Active Threads. Instead, Active Threads supports a general scheduling event mechanism that enables many different extensible scheduling policies. Each bundle can implement its own scheduling policy independent of the rest of the system. The policy is implemented by the *bundle scheduler.* The Active Threads scheduling mechanism defines interfaces to which all such schedulers must conform. The Active Threads distribution comes with a library of common schedulers which can be easily extended to fulfill unforeseen needs.

Event Mechanism

Active Threads provides no hard-coded scheduling policy. Instead, it supports a general mechanism upon which different specialized scheduling policies can be built.

All thread scheduling decisions are made by the bundle to which the thread belongs. The bundle is free to maintain any scheduling data structures that fit most closely the semantics of the thread group. Such an architecture enables the coexistence of various scheduling policies within a single application and facilitates the encapsulation of thread scheduling. A parallel library module is envisioned as consisting of data structure sources and the associated schedulers used together.

Different susbsytems communicate with bundles by vectoring scheduling events. Bundles encapsulate all aspects of scheduling and must provide event handlers for all scheduling events. There are no restrictions on the implementation of such event handlers.

The relationship between bundles and other subsystems as well as the direction of the event flow are shown in Figure 5.

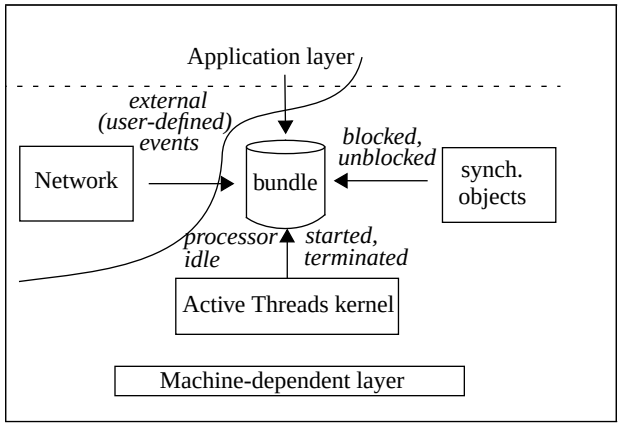

Figure 5: Scheduling events in Active Threads

All events are partitioned into two groups: internal Active Threads events and external (user-defined) events. Internal events deal with common thread operations: thread creation, termination, blocking, unblocking, dispatching by the processor, etc. External events facilitate the implementation of less common and more specialized thread operations such as thread migration in distributed systems or committing threads to a persistent store. In this section, we will concentrate on the internal events. The details of how new Active Threads events can be defined to implement thread migration in a network of SMPs as well as an involved performance analysis of thread migration are given in [65].

All internal events, their logical origins and a short description of the information they provide to the thread bundle are given in Table 2

thread created (kernel)	informs about creation of a new thread.
thread started (kernel)	informs about thread start-up; enables lazy stack allocation policies.
thread terminated (kernel)	informs about thread termination.
thread blocked (synch. object)	informs about thread blocking on a synchronization object
thread unblocked (synch.objects)	informs about thread unblocking.
bundle created (kernel)	informs about creation of a new child bundle.
bundle terminated (kernel)	informs about termination of a child bundle.
processor idle (kernel)	requests more threads for dispatching by the idle processor.

Table 2: Internal events.

Normally, threads in the ACTIVE (executing) state continue running until they block on synchronization objects, yield execution or terminate. When one of these conditions occurs, the runtime performs the associated action which involves only a limited number of steps. For instance, when a thread blocks or yields, the thread needs to be stopped and its context needs to be saved in the thread control block. This involves saving the contents of a portion of a register file in a place provided by the thread data structure. Which registers need saving varies across different architectures and compiler parameter passing conventions. When a thread terminates, associated data structures such as the thread control block and stack are returned by the runtime to corresponding pools that it maintains. After the elementary thread operations are completed, a thread event that contains a thread handle is dispatched to an appropriate bundle.

In practice, the *thread created* and *thread unblocked* events can be handled by a single handler. Schedulers that choose to implement lazy stack allocation to reduce overall memory consumption can use the *thread started* event to create a thread's stack on thread startup rather than thread creation (we will show in later sections that this may substantially reduce maximum memory requirements). Unless a bundle wants to keep an exact account of which threads are blocked and running, it can supply a null *thread blocked* and *thread terminated* event handlers. Bundles implementing standard eager stack allocation may also supply a null *thread started* event handler. In most of our experiments, bundles provided distinct event handlers for only four events: *thread created*, *bundle created*, *bundle terminated* and *processor idle*. Lazy stack allocation adds a handler for the *thread started* event. All scheduling event handlers are necessary to implement complex scheduling policies such as priority scheduling with aging.

As one example of the use of scheduling events, Figure 6 illustrates what happens when a thread blocks on the synchronization objects and later unblocks.

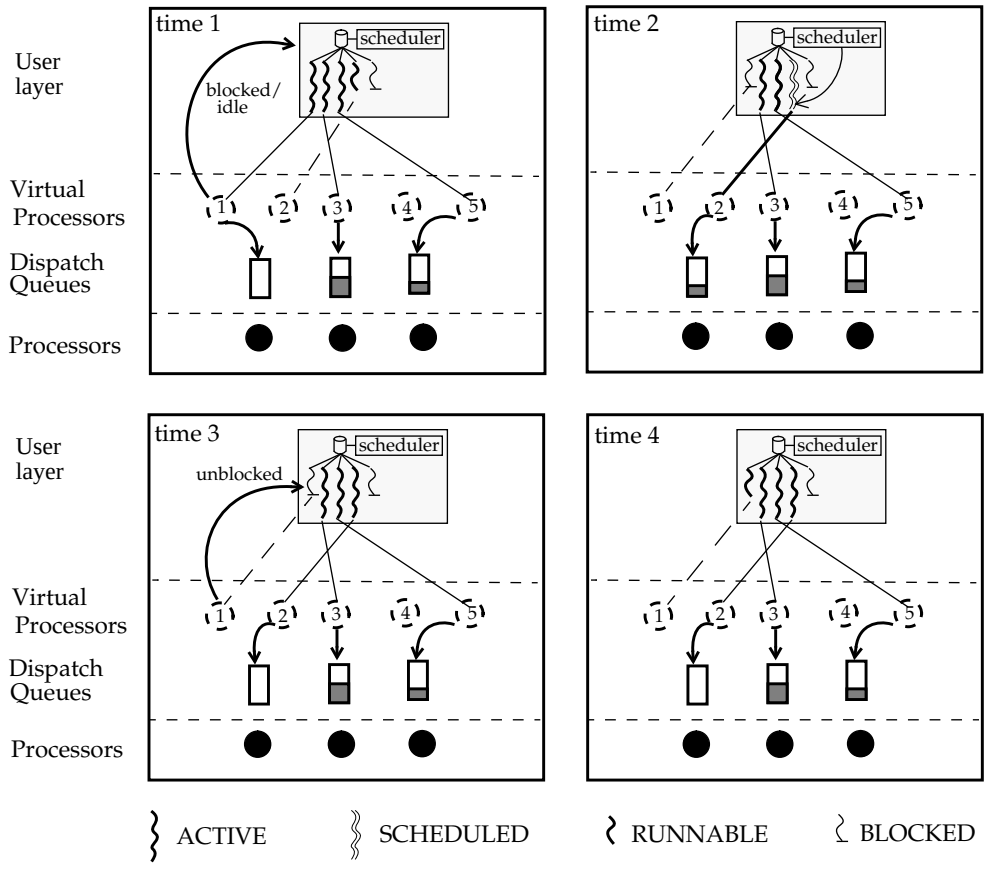

Figure 6: Example of a scheduler event mechanism

Figure 6 is simplified in several minor respects - for instance, kernel threads are not even displayed - but it captures the essence of the Active Threads event mechanism. At time 1 a thread executing on virtual processor 1 blocks. A thread *blocked* event is generated to communicate this information to the bundle. As the figure shows, the physical processor executing a blocking thread has an empty thread dispatch buffer. When a thread blocks, a physical processor becomes idle and an *idle* event is vectored to the bundle to request more work.

At time 2, the bundle has processed both events. Internal bundle scheduler data structures have been updated to reflect blocking of a thread. In response to an *idle* event, a bundle selects an available thread in the RUNNABLE state for execution on virtual processor 2. The selected thread switches to the SCHEDULED state as shown in Figure 6 for time 2. It is subsequently dispatched by the idle physical processor to become ACTIVE.

At time 3, a synchronization object receives a signal and unblocks the previously blocked thread. A thread *unblocked* event is generated by the Active Threads runtime and gets forwarded to the bundle of the unblocking thread. The bundle updates the associated data structures to reflect that the thread is unblocked. The unblocked thread becomes RUNNABLE (time 4).

Our example in Figure 6 was simplified for clarity. In the example, only a single bundle is shown. There may be many concurrent bundles utilizing different schedulers. This creates no ambiguity with respect to thread and bundle events. Since there is a unique mapping from threads and bundles to their parent bundle in the activation tree, thread and bundle events are forwarded to such parent bundles. However, when a processor becomes idle, it needs to select a destination bundle for an *idle* event. One way to solve this problem is to always forward *idle* events to the root of the bundle activation tree and then rely on bundle to propagate the event to the right place in a tree. For large activation trees, the overhead of such propagation (which may involve the traversal of the entire bundle activation tree in some order) can easily surpass the actual context switch time and become a scheduling bottleneck.

For this reason, we have chosen a solution similar to that used in TAM [9][46][13] for non-blocking threads. In TAM, all threads are organized in groups called code blocks. A runtime representation of a code block is called an activation. Activations form a tree hierarchy not unlike that of the bundle activation tree in Active Threads. There may be many activations present in the system, but a processor can execute threads only from *resident* activations. The compiler explicitly emits code to make activations resident. Once an activation becomes resident, it remains so until all its threads terminate. This is an exceptionally good model that promotes locality for lenient programming languages with unblocking underlying threads. We have extended these ideas to provide a mechanism that supports scheduling of blocking threads.

At any moment, a unique bundle is said to have the execution *focus*. Active Threads API provides a primitive for setting and obtaining a value of a focus bundle. Whenever processors become idle, they send *idle* events to the bundle with the focus. Similar to a resident activation in TAM, a focus bundle serves as a work producer for physical processors. However, in Active Threads, scheduling does not need to be centralized physically. It is up to the focus bundle to schedule more threads or to propagate the *idle* event up or down the bundle activation tree. In the presence of enough parallelism, a single bundle will be allocated all resources. Simple schedulers that facilitate cache reuse can be used in such settings. When a single bundle cannot keep all processors running, a more complex protocol needs to be followed to propagate scheduling requests to other parts of the bundle activation tree. Such organization allows the implementation of scheduling with different degree of centralization: from completely centralized with all threads managed by a single bundle to a completely distributed. Section 5.4 provides some code examples that illustrates these points.

The scheduling event mechanism is somewhat reminiscent of the *scheduler activations* mechanism intended to bridge the gap between user level threads and OS kernel in the presence of I/O [1]. Active Threads events, however, fulfill different goals - transparent composition of lightweight schedulers that may take advantage of temporal and spatial locality.

Scheduling policies

While Active Threads provides a general scheduling mechanism, bundles implement different scheduling policies. Although no restrictions are imposed on the algorithms and data structures used to implement scheduling policies, the following general principles were set forth to guide the design and implementation of standard bundle schedulers distributed with Active Threads.

- **Using locality information**. Regardless of the scheduling details, bundle are encouraged to use thread affinity information.
- **Minimizing rescheduling overhead**. Fine-grain parallel programming imposes restrictions on system overhead. It is possible to perform a thread context switch in just a few dozen instructions on a wide range of modern architectures (section 7). Rescheduling overhead must be equally low to keep threads lightweight.
- **Minimizing hot spots**. Scheduler data structures should be carefully chosen to avoid hot spots. Contention for access to common resources will have a negative impact on fine-grain parallel applications.

The Active Threads distribution comes with library of commonly used schedulers (section 5.4).

3.5 Synchronization Objects

The Active Threads scheduling events mechanism enables creation of synchronization objects with different synchronization policies. Many commonly used synchronization objects are provided by the Active Threads library:

- spinlocks: simple, snooping and exponential back-off
- mutexes (two-phase blocking mutual exclusion locks)
- reader/writer locks
- semaphores
- condition variables

Active Threads also supports atomic acquisition of sets of locks. Full interfaces for the synchronization objects supported by the Active Threads library are presented in the Appendix. Performance and implementation issues in atomic acquisition of sets of locks in Active Threads are explored in [43]

While many common synchronization patterns are captured in the Active Threads library, the scheduling event mechanism supports some unforeseen extensions. For instance, a barrier lock is not supported directly by Active Threads, but could be easily implemented using scheduler events.

4 PROGRAMMING EXAMPLE

This section presents a simple programming example that emphasizes the importance of preserving temporal and spatial locality for performance. The example is very simple yet it is capable of illustrating many concepts presented in the previous section. However, it should not be perceived as a tuned algorithm that exploits locality to the fullest. Better CPU utilization could be achieved by various blocking strategies at different levels of memory hierarchy, but similar ideas apply even to more complex and efficient implementations.

Section 4.1 examines a simple parallel version of vector-matrix multiplication to illustrate the Active Threads API. Section 4.2 shows how a minimal change to the naive implementation that takes into account spatial locality can have substantial performance implications.

4.1 Example of API Usage: Vector-Matrix Multiplication

We have selected vector-matrix multiplication to illustrate the Active Threads API. There are several reason for this choice:

- Simplicity.
- Natural decomposition into fine-grain units of computation that can be performed in parallel. Threads are a natural way to express this decomposition. This also allows the implementation to avoid dependencies on the number of physical processors.
- Fine-grain parallelism imposes strict performance constraints on the implementation of the underlying parallel runtime system. Thread management overhead limits the granularity of parallelism and to achieve speedups, key thread operations must be extremely lightweight.
- In spite of its simplicity, the problem illustrates the benefits of exploiting spatial and temporal locality.

A segment of code that implements simple parallel vector-matrix multiplication is shown in Figure 7.

```
/* Multiply vector 'v[rows]' by matrix 'm[rows][cols]' and keep the result in 'r[cols]' */
void vxm(float *v, float *mat, float *r, int rows, int cols){
    int i,col;
    at_sema_t *sema = at_create_sema(0); /* semaphore signaling completion */
    /* create a thread for each vector - matrix column product */
    for(col=0; col<cols; col++){
        /* create an unbound thread and attach it to a bundle with execution focus */
        at_create_6(at_get_focus(), AT_UNBOUND, vxcol, v, mat+col, rows, cols, &r[col],
                    sema);
    }
    /* Wait until all dot products terminate */
    for(i=0; i<cols; i++) at_sema_wait(sema);
    am_sema_destroy(sema);
}

/* compute a dot product of vector v[n] and column 'col' of matrix mat */
/* The column is specified by a pointer to the first element e1 and 'stride' */
void vxcol(float *v, float *e1, int n, int stride, float *res, at_sema_t *sema){
    int i;
    *res =0;
    for(i=0; i<n; i++)
        res += v[i]*(*(e1+stride*i));
    at_sema_signal(sema); /* signal completion */
}
```

Figure 7: Vector-matrix multiplication

The computation naturally decomposes into dot products of the input vector and matrix columns. A separate thread is created for each such dot product. New threads are added to the currently active bundle. The AT_UNBOUND annotation specifies that a thread has no affinity for any processor. A simple semaphore is used to signal completion of dot products to the parent thread. The parent thread is blocked on the semaphore until all dot products have terminated and the resulting vector is fully computed.

Code in Figure 7 is fairly straightforward and can be mechanically converted to use, for instance, POSIX or Solaris threads instead of Active Threads.

4.2 Memory-Conscious Thread Scheduling and Performance

The code segment in Figure 7 is extremely simple, but it fails to address memory locality issues. In particular, it displays very poor spatial locality. Data items that are located close in space are accessed by different threads. As a result, the parallel version has a speedup of about 3 on the 8 processor Sun Enterprise 5000 (each processor is a 167Mhz UltraSPARC-1). While better speedups could be achieved by parallelizing a different (serial) algorithm tuned to memory hierarchy, simple processor affinity annotations allow us to obtain similar results without increasing code complexity.

First, we examine some problems of our naive implementation. Consider a thread that multiplies vector v by matrix column j. A physical processor that executes this thread will end up loading column j of matrix m in its cache. For simplicity, we consider only the case

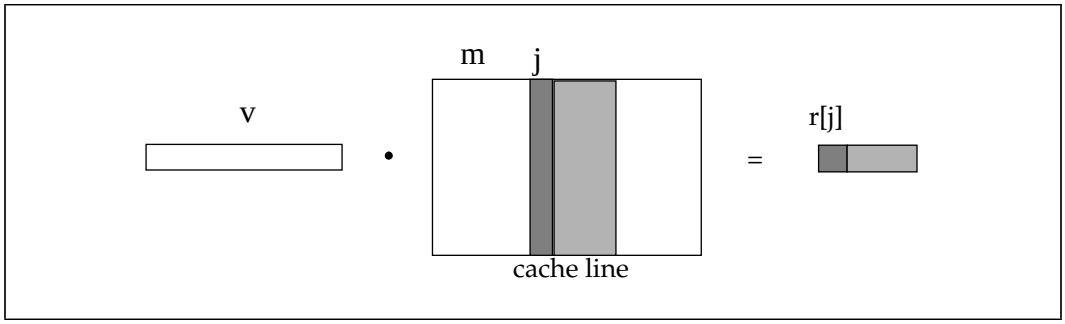

Figure 8: Vector-matrix multiplication and cache effects

of a single level cache. All issues considered here apply to architectures with multiple level caches. We also assume that all necessary data fits in cache entirely.

Figure 8 displays the portion of matrix m that ends up in cache of the considered processor by the time the dot product thread terminates. Caches of modern machines are usually organized as some number of cache lines with cache lines of 32 or 64 bytes being fairly common. The entire line is loaded into cache on reference to a matrix element. On the muliprocessor, this can be in conflict with work partitioning and thread scheduling. The naive implementation of matrix-vector multiplication suffers from such a conflict.

The failure to take memory hierarchy into account results in the following problems:

- The total number of cache misses necessary to load data increases. For example, a thread that computes a dot product of vector v and matrix column $j+1$ can execute on any processor and will most likely load column j+1 into cache again (it is already in cache of our original processor).
- There are additional misses due to false sharing. For instance, if threads that use columns j and $j+1$ execute concurrently, a thread storing a dot product result in $r[j]$ causes cache invalidation for the processor that computes $r[j+1]$. Invalidation happens even though the processors do not write to the same location.
- Extensive cache traffic due to the avalanche of cache misses saturates the bus and imposes a limit on the number of processors that can be used effectively. For instance, increasing the number of processors beyond 5 for our naive implementation is of little benefit. A saturated bus simply cannot supply enough data to keep all pr ocessors running efficiently . The speedup curve stabilizes at around 3 even if we keep increasing the number of processors (Figure 10).

The virtual processor annotation mechanism of Active Threads allows the user to eliminate most of excess cache traffic in a simple and portable manner. Minimal modifications to the original vector-matrix multiplication deliver competitive performance even for our naive fine-grain threaded implementation. An updated version is shown in Figure 9 with added lines marked /* new */.

```
/* Multiply vector 'v[rows]' by matrix 'm[rows][cols]' and keep the result in 'r[cols]' */
void vxm(float *v, float *mat], float *r, int rows, int cols]){
    int i,col;
    at_sema_t *sema = at_create_sema(0); /* semaphore signaling completion */
    at_bundle_t *bundle = at_mcs_bundle_create(); /* new */
    /* create a thread for each vector - matrix column product */
    for(col=0; col<cols; col++){
        int vproc = col/64; /*new */
        /* create an unbound thread and attach it to a bundle with execution focus */
        at_create_6(bundle, vproc, vxcol, v, mat+col, rows, cols, &r[col],
                    sema); /*modified */
    }
    /* Wait until all dot products terminate */
    for(i=0; i<cols; i++) at_sema_wait(sema);
    am_sema_destroy(sema);
    at_bundle_destroy(bundle); /* new */
}
```

Figure 9: Vector-matrix multiplication with processor affinity annotations

This version creates a new bundle that supports scheduling with respect to virtual processor annotations. Threads that access neighbor columns of the input matrix m are scheduled to run on the same virtual processor. The Active Threads runtime takes care of mapping virtual processors to physical processors and load balancing. Figure 10 presents performance results of the original and modified versions. Replacing the thread scheduler leads to significant performance gains and much higher speedups.

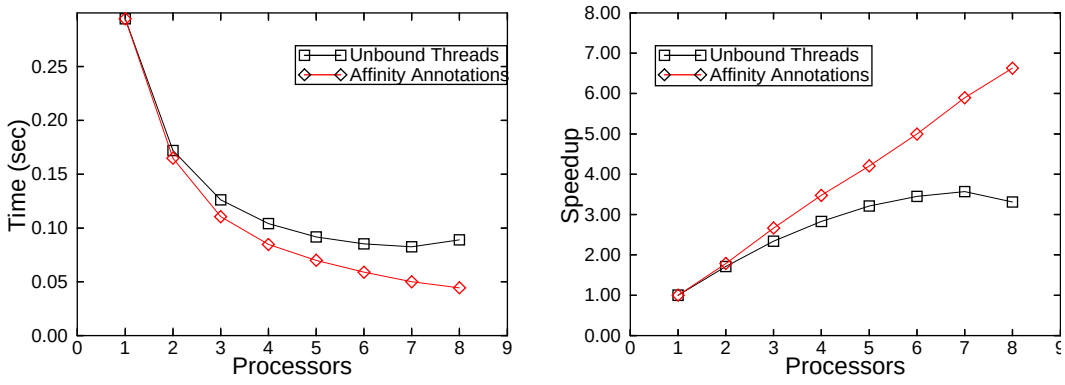

Figure 10: Vector-matrix multiplication performance

5 IMPLEMENTATION

5.1 Overview

The Active Threads system consists of five main units: *Active Threads Kernel*, *Synchronization Objects*, *Schedulers*, *Memory Management*, and *Machine-Dependent Layer*. Active Threads units and relationships between them are shown in Figure 11.

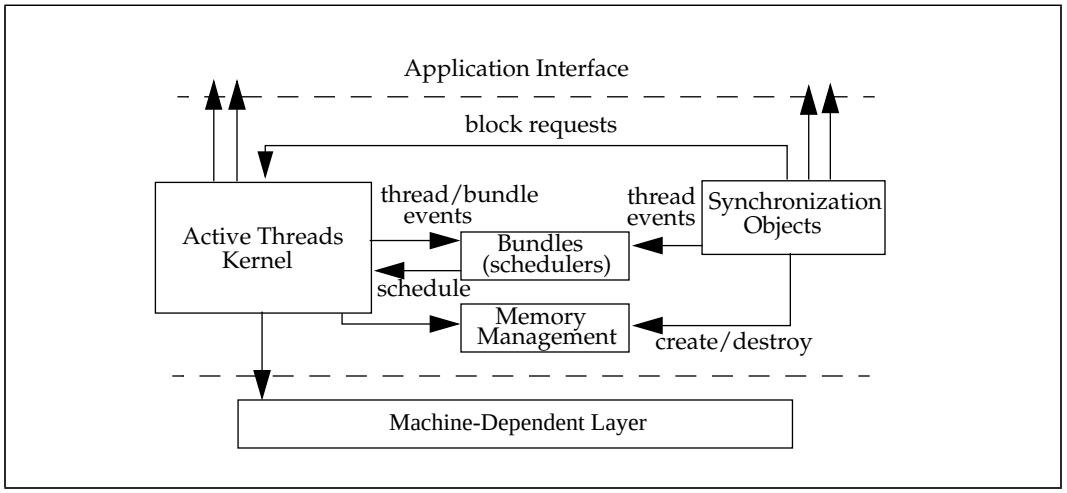

Figure 11: Active Threads Subsystems

All subsystems other than the machine-dependent layer are platform independent. The Active Threads kernel provides the basic thread interface: thread creation and termination, thread startup and concurrent execution. Active Threads synchronization objects capture different synchronization patterns such as spinning and blocking mutual exclusion, reader/writer locks, counting semaphores, condition variables, etc. New kinds of synchronization objects can be added with other system units intact. Bundles implement different kinds of scheduling policies for collections of threads. As in the case of synchronization objects, new kinds of schedulers can be added to the system with no modification to other subsystems. The memory management unit provides parallel memory management for all internal purposes. The machine-dependent unit captures all architectural dependencies.

We now discuss the Active Threads susbsytems in more detail.

5.2 Active Threads Kernel

The kernel unit provides facilities for thread and bundle creation, termination, startup, context switch, and thread stack allocation. The kernel unit is the only Active Threads unit that communicates with the machine-dependent layer to request the low-level architecture specific services. All other Active Thr eads subsystems rely on the kernel for basic thread operations.

Each thread's state information is stored in the *thread context block*. The register state on context switch is stored on the thread stack. The pointer to this area is in the thread context block. The context block also keeps other miscellaneous thread information and a pointer to a thread bundle.

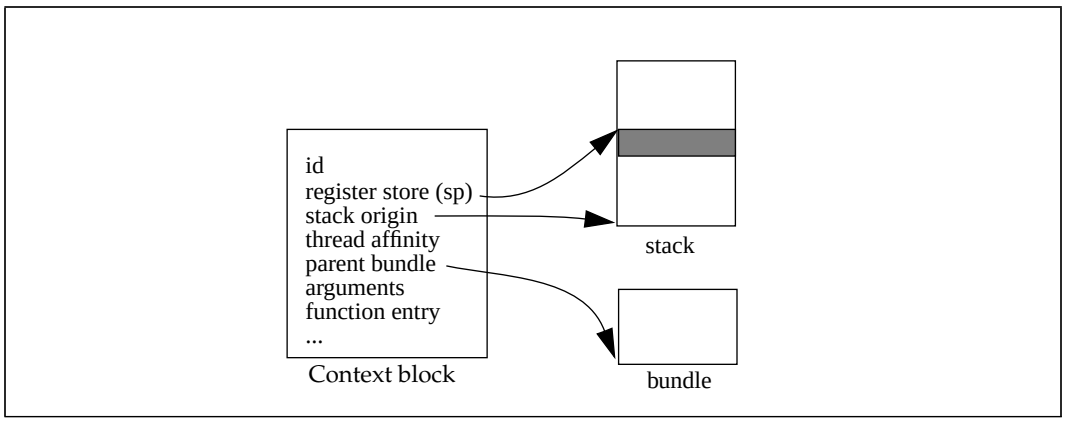

Figure 12: Thread context block.

Each thread has a unique id which can be obtained by calling *at_self()*. Each thread runs with its own stack. Thread context block and stack allocation and deallocation are handled by the memory management unit. However, the binding of threads to stacks is performed by the bundle. The time of such binding may vary. We will show in section 8.1 that late binding of threads to stacks can significantly reduce overall memory requirements. Bundles may choose to implement different binding policies.

Thread affinity information stored in the context block is used by bundles that support locality scheduling to influence the assignment of threads to physical processors.

When a thread is created, the kernel obtains a fresh thread context block from the memory management unit, initializes it and vectors a *thread created* event to an appropriate bundle. When a thread is scheduled to run for the first time, a *thread started* event is forwarded to the bundle. If the bundle implements lazy stack allocation, a thread is bound to a stack by the *thread created* event handler. On thread termination, the kernel vectors a *thread terminated* event to the thread's bundle. Similar events are associated with bundle creation and termination operations. The Active Threads kernel is also responsible for generating *processor idle* events when processors run out of work.

Processor Dispatch Queues

The Active Thread kernel maintains *thread dispatch queues* (or buffers) for each processor. Bundles deposit threads in the dispatch queues in response to scheduling events.

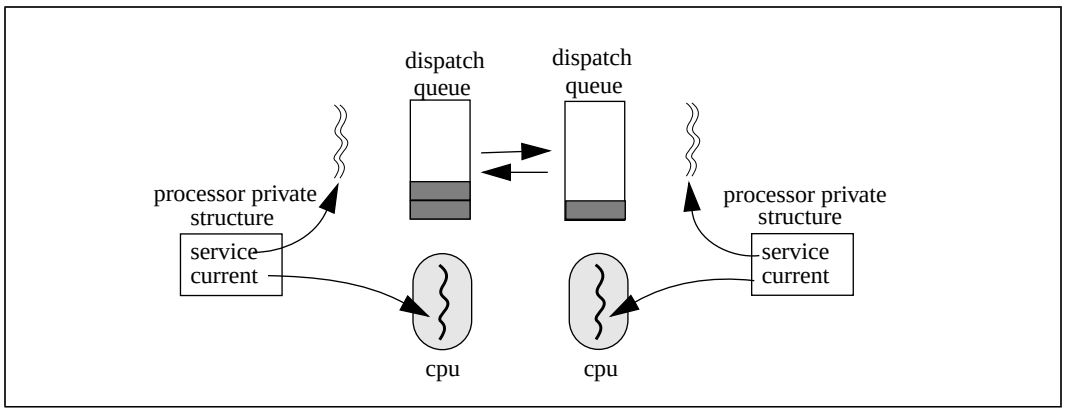

Figure 13: Processor dispatch queues

Normally, when a thread blocks, the associated processor picks up a runnable thread at the head of a its dispatch queue. The code executed on behalf of the new thread (getting the thread from the dispatch queue) is executed using the stack of the blocking queue to avoid an extra context switch. The alternative of using a special scheduling thread for this purpose imposes an extra context switch to a scheduling thread that consults the dispatch buffer for each thread blocking operation.

If the dispatch queue is empty, an *idle* event is generated. If the bundle that handles this event deposits some thread in the dispatch buffer, the context switch completes as above. However, if the dispatch buffer remains empty, as a final resort, dispatch buffers of other processors are consulted.

Finally, if no new threads can be obtained, a special service thread wakes up. It is responsible for requesting more work by vectoring *processor idle* events to the active bundle. As long as the dispatch buffers are kept non-empty, the service threads are asleep. A service threads only wakes up when a processor is idle and no new threads are scheduled in response to the *processor idle* event. In the current implementation, the service thread keeps control over the processor even if no scheduled threads are available. We plan to amend this situation by releasing the processor in the future versions of Active Threads to improve throughputs in multiprogramming environments. Another related extension is either switching to a new thread or releasing a processor when a currently active thread is blocked in the OS kernel on I/O. However, this requires support services from the OS kernel such as scheduler activations [1].

Unlike other threads systems such as PRESTO [3], Active Threads does not use a special *scheduler* thread to manage context switches and rescheduling. As mentioned earlier, such architecture requires an additional context switch to the scheduler thread for each blocking operation. Instead, Active Threads implements the concept of a *preswitch* [27]. After the old thread is blocked, the processor switches to the stack of a new thread and dispatches the *thread blocked* event to the active bundle. The bundle may run some code on behalf of the old thread in the stack of the new thread and the extra context switch is avoided.

The Active Thread kernel also makes sure the new thread indeed has a stack. A race condition may arise when a lazy stack allocation policy is used. If the new thread is scheduled to run for the first time, the kernel vectors a *thread started* event to the active bundle. Any valid bundle implementation must guarantee that after this event is handled, a stack for the new thread has been created. Some examples of lazy stack allocation are given in section 5.4.

A processor private structure (shown in Figure 13) is allocated per processor and is used to keep pointers to the context blocks of the currently active thread and the service thread for that processor. The processor private structure also contains some other miscellaneous processor specific information. Locating a processor private structure by the cpu may present a problem if the processor does not support at least one word of private memory. Since each processor uses a different private structure, pointers to such structures cannot be located in the ordinary RAM - different processors share text and data segments and will inadvertently share the private structure. On many architectures the problem is resolved by using processor-specific memory. For instance, the SPARC architecture [52] reserves a global register *g7* as processor-private storage. The SPARC parameter passing convention ensures that compilers and operating systems do not use this register - it is intended to be controlled exclusively by the thread system [47]. If a processor does not support any private storage, the current value of a stack pointer can be used to infer the location of the processor private storage (section 5.6).

5.3 Synchronization Objects

Active Threads currently provides several kinds of synchronization objects:

- spinlocks: simple, snooping and exponential back-off
- blocking (two-phase) mutual exclusion locks
- blocking reader/writer locks
- counting semaphores
- barriers
- condition variables

The APIs of the above synchronization objects are presented in the Appendix. The standard Active Threads synchronization objects have conventional semantics and essentially match or supersede the functionality of POSIX or Solaris Threads.

New kinds of synchronization objects can be added with no modification to other functional units. To illustrate the flexibility of the Active Threads event mechanism and extensibility of synchronization objects, we present the code for the mutex unlock operation (Figure 14)

```
void at_mutex_unlock(at_mutex_t *m){
    at_thread_t *t=NULL;
    at_bundle_t *b;
    AT_SPINLOCK_LOCK(m->slck); /* protect sleeping queue updates */
    m->owner = NULL; /* clear the mutex */
    if(m->sleepers){ /* select a thread to wake up, if any */
        t = m->sleepers;
        m->sleepers = t->next;
    }
    AT_SPINLOCK_UNLOCK(m->slck);
    if(t){
        b = t->bundle;
        /* dispatch a "thread unblocked" event to bundle b*/
        AT_EVENT(b, thread_unblocked, t);
    }
    return;
}
```

Figure 14: Mutex unlock operation.

Synchronization objects are free to use any representation for queued sleeping threads. Synchronization objects communicate with the rest of the system by vectoring scheduling events to bundles and calling the kernel-provided *at_block()* primitive to safely block running threads. In Figure 14, sleeping threads are kept in a simple FIFO queue. If the queue is non-empty, the thread at the head of the sleeping queue is removed and a *thread unblocked* event is generated to inform the thread's bundle. Figure 15 illustrates how a bundle that supports a simple FIFO policy catches this event and immediately schedules the unblocking thread for execution. More complex priority-based mutual exclusion locks can be implemented by selecting the appropriate representation for the sleeping queues and modifying the selection criteria.

```
void thread_unblocked(at_bundle_t *b, at_thread_t *t) {
/* a simple event handler for bundle 'b' that just schedules the thread */
    at_schedule(t);
}
```

Figure 15: A simple *thread unblocked* event handler.

5.4 Extensible Bundle Schedulers

An Active Thread bundle must implement event handlers for all eight internal scheduling events (section 3.4). No restrictions are placed on the bundle internals. The implementations are free to select the best data structures possible to implement the scheduler. For instance, if the number of threads is known statically (or even dynamically at bundle creation

time), the scheduler may keep threads in an array to minimize rescheduling overhead. In simple dynamic cases, a linked list may be sufficient. However, to implement priority scheduling, more complex (and expensive) data structures such as priority queues are used. In general, the least expensive bundle that fully implements the desired functionality should be chosen.

The Active Threads package (version 1.2) is distributed with a library of bundles that implement the following commonly used schedulers:

- FIFO
- FIFO with memory-conscious scheduling (MCS)
- FIFO with lazy thread stack allocation
- FIFO with lazy stack allocation and MCS
- LIFO
- LIFO with MCS
- LIFO with lazy thread stack allocation
- LIFO with lazy stack allocation and MCS

We plan to extend the library by adding several non-preemptive priority-based schedulers. We also intend to add bundles that can utilize the hardware performance counters present in many modern architectures [54][41].

To illustrate how easily new bundles can be created and added to the system, we will show how a simple FIFO bundle can be modified to implement a lazy thr ead stack allocation policy. The benefits of the lazy stack allocation will be demonstrated in section 8.1. Lazy task creation has been discussed in the literature and some language runtime systems are exclusively based on this strategy [38][14][15]. Lazy stack allocation is somewhat similar to these systems in that many threads can potentially execute using a single physical stack.

A bundle that implements FIFO scheduling with lazy stack allocation shares most scheduling event handlers with a simple FIFO bundle. Only the *thread created* and *thread started* event handlers are different.

```
/* thread created event handler */
void fifo_thread_created(at_bundle_t *b, at_thread_t *t){
    at_create_stack(t);          /* Allocate a new thread stack */
    at_create_local(t);          /* Create thread local storage if needed */

    /* add to the internally kept FIFO list*/
    fifo_bundle_add_thread(b, t);
}

/* thread started handler - nothing to do */
void fifo_thread_started(at_bundle_t *b, at_thread_t *t){
}
```

Figure 16: Original FIFO event handlers

Figure 16 presents the original code for the two event handlers. The handlers are extremely simple. The *thread created* handler first allocates a stack and thread-local memory for a newly created thread. This is done by calling the corresponding functions provided by the Active Threads kernel interface. Then, the thread is added to the queue maintained by the FIFO bundle. The enqueued thread is fully initialized and is ready for execution. The *thread started* event handler remains empty.

```
/* thread created event handler */
void fifo_lazy_thread_created(at_bundle_t *b, at_thread_t *t){
    /* stack and thread local memory are allocated at thread startup */
    /* simply enqueue the new thread on the bundle thread list*/
    fifo_lazy_bundle_add_thread(b, t);
}

/* thread started handler - has to allocate the stack and local memory */
void fifo_lazy_thread_started(at_bundle_t *b, at_thread_t *t){
    at_create_stack(t);          /* Allocate a new thread stack */
    at_create_local(t);          /* Create thread local storage if needed */
}
```

Figure 17: FIFO with lazy stack allocation

Figure 17 shows how the original code is modified to obtain lazy stack allocation semantics. All that is different is that the new stack and thread local memory are allocated at thread startup rather than the thread creation time.

We will show in section 8 that lazy allocation can dramatically reduce run time and memory requirements. For instance, for non-blocking threads and a lazy stack allocation policy, the total number of used physical stacks is never greater than the number of processors.

5.5 Memory Management

To decrease allocation latencies, Active Threads provides extensive pooling for all allocated entities: thread context blocks, stacks, synchronization objects, etc. A single parallel pool data structure is used in all cases, however different pool instances are created for different kinds of objects. Because Active Threads internally uses s fixed set of objects with statically known sizes, we can achieve lower allocation latencies by not resorting to general purpose allocators [51].

A parallel pool utilized by Active Threads satisfies all allocation requests. It maintains local pools organized as linked lists for each cpu. Normally, when a processor requests a new object, the request is satisfied from a local pool transparently to other processors. Similarly, objects that are no longer needed are returned to the local pools of the cpus where they were used last. Local pools are implemented as LIFO queues to maximize cache reuse - objects at the head of the queues are likely to be cached by processors that issue allocation requests.

If a request cannot be satisfied from a local pool, other processors' pools are examined. If a pool with a sufficiently large number of objects is found, half of its elements are transferred to an empty pool. Finally, if no such pools are available, a new chunk of objects is allocated.

The parallel pool data structure currently performs no attempt to pad allocated objects in order to eliminate cache traffic to avoid multiple objects spanning cache lines. Such optimization may be of particular importance for synchronization objects and will be added in future releases.

5.6 Machine-Dependent Layer: Portability

The machine-dependent layer hides the hardware details of modern multiprocessors. The Active Threads kernel never addresses the underlying hardware directly: all needed services are performed by the machine-dependent layer. Porting Active Threads to a different architecture involves only retargeting the machine-dependent layer. The machine-dependent layer is ported by implementing the Active Threads Portability Interface on the intended platforms.

Portability Interface

Active Threads has been ported to a variety of platforms including SPARC and Intel i386 and higher running Solaris, DEC Alpha AXP running OSF and HPPA running HPUX. The implementations of the Portability Interface are on the order of a hundred C and assembly lines and were performed in a matter of days.

Primitive	**Semantics**	
initialize	initialize the machine-dependent subsystem	a few dozen C lines
has-private-storage?	true if the platform supports a single word of a processor-private storage	1 C line
start light-weight process	start a new light-weight process executing a scheduling loop	a dozen C lines
light-weight process set private	set a single word of light-weight process specific storage	1-3 assembly lines
light-weight process get private	obtain a word of a light-weight private storage	1-3 assembly lines
read-and-modify, memory value	an atomic memory read and modify operation and a memory value after modification.	a few assembly lines
number of physical processors	return the number of currently available physical processors	a dozen C lines
thread context switch	switch between two thread contexts. Involves stopping the thread, and saving and reloading registers.	a few dozen assembly lines
thread initialize	initialize the stopped thread's context with a function address and arguments.	a few assembly lines

Table 3: Active Threads Portability Interface

The *initialize* primitive is guaranteed to be called by the Active Threads runtime before any other Portability Interface primitives are invoked. The *has-private-storage* macro is set if the platform supports at least one word of private per-processor private storage. On machines with per-processor private memory, it could be used to store the address of a processor-specific data structure that contains different data on different processors. The details of such data structures were described earlier in section 5. Among other things, processor private structures contain a handle for a currently executing thread.

On machines that support processor private storage, setting and obtaining the private data can be done very efficiently - it only involves a single load or store instruction. For instance, the SPARC architecture [52] reserves a global register *g7* as processor-private storage. The SPARC parameter passing convention ensures that compilers and operating systems do not use it for other purposes [47]. The benefits of processor private storage for threaded parallel runtime systems are father discussed in [27].

If the architecure does not support per-processor memory, Active Threads uses the content of the stack pointer to obtain a handle for processor-specific data structures as shown in Figure 18.

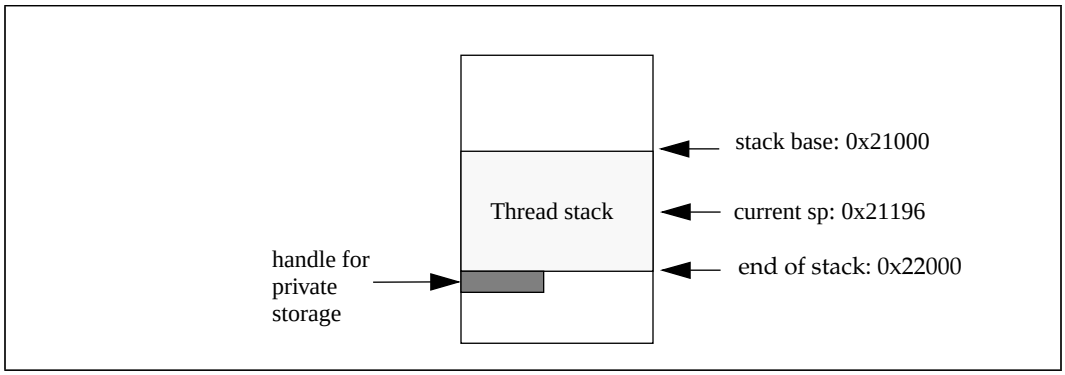

Figure 18: Using the stack pointer to find processor-specific data structures.

The idea is to always align thread stacks on the stack size boundary. For instance, in Figure 18, the base of a stack of size 0x1000 is at address 0x21000. A handle of a per processor storage (or even the processor private data structure) can be placed immediately following the stack. Then, given any valid value of a running thread's stack pointer, we can simply and efficiently compute the address of a processor private handle:

(sp&~stack_size)+stack_size+word_size.

The implementation of the context switch is based on the earlier work by David Keppel [27]. The context switch is done in just a few dozen instructions on all supported architectures.

6 FUNCTIONALITY COMPARISONS

In this section, we compare the functionality of Active Threads with that of the three commercial general-purpose thread systems, the POSIX Threads standard and the Java Threads specification. The purpose of this comparison is to demonstrate that while Active Threads has somewhat distinct goals from the general-purpose commercial thread packages and delivers substantially higher performance, Active Threads is comparable in expressiveness of basic thread operations with the other systems. In addition, Active Threads provides a unique array of services not found anywhere else.Table 4 is partially based on [28] and [40].

In Table 4, "yes" means that the functionality is directly supported by the thread system. "buildable" means that the functionality, although not directly supported by the system, can be easily constructed. "difficult" means that although it may be possible to construct the desired functionality, it may be very labor-intensive with possibly negative performance implications. It may not be possibly to build all the desired functionality. "impossible" means the functionality cannot be constructed by relying on the system provided primitives [28].

Functionality	Solaris Threads	POSIX Threads	NT Threads	OS/2 Threads	Java Threads	Active Threads
user/ kernel level	user	user or kernel	kernel	kernel	user or kernel	user
time-slicing	possible	possible	yes	yes	undecided	no
scheduling	global priority	several classes	global priority	global priority	global priority	composi-tional
scheduling extensibility	difficult	difficult	difficult	difficult	difficult	yes
synchronization extensibility	difficult	difficult	difficult	difficult	difficult	yes
mutexes	yes	yes	yes	yes	buildable	yes
semaphores	yes	yes	yes	yes	buildable	yes
R/W locks	yes	buildable	buildable	buildable	buildable	yes
condition variables	yes	yes	impossible	impossible	yes	yes
multi-object synchronization	difficult	difficult	yes	yes	difficult	yes
thread suspension	yes	impossible	yes	impossible	yes	no
thread-specific data	yes	yes	yes	difficult	buildable	yes
thread signals	yes	yes	n/a	n/a	n/a	n/a
no compiler changes required	yes	yes	no	yes	n/a	yes
portability	unknown	n/a	unknown	unknown	n/a	yes

Table 4: Thread systems functionality.

Both NT and OS/2 provide kernel-level threads. The POSIX specification allows both user-level and kernel-level implementations. There is a trend to base Java Threads on the native threads provided by the platforms and hence there are kernel-level and user-level implementations of Java Threads.

Both Solaris and POSIX Threads provide time-slicing for special kinds of scheduling classes, while the default behavior does not involve preemption. The Java language specifications is ambiguous about time-slicing in Java Threads. Some implementations of Java Threads, such as the NT version are time-sliced, while most implementations for Unix platforms are not.

Active Threads does not support time-slicing because of performance and portability implications. Negative performance implications of time-slicing are well known [2][27]. Time-slicing also introduces a dependency on the OS kernel for a timer interrupt signal to initiate a context switch, which presents a portability challenge.

Most thread systems provide a single scheduling policy based on global priorities. For instance, Java Threads support only 10 different priorities [40]. POSIX threads can be configured to use one of the several predefined scheduling policies such as priority or FIFO. Active Threads is the only system that provides compositional scheduling with many extensible scheduling policies. No other thread systems provide a mechanism for addition of new scheduling policies. In some cases, the desired functionality can be achieved by building a new scheduler on top of a system provided priority scheduler with the help of synchronization objects. This approach is both complex and unlikely to yield good performance.

Similarly, Active Threads is the only system with extensible synchronization objects. In all other systems, new synchronization objects can only be built on top of the provided primitive objects such as mutexes, semaphores, etc. Neither NT nor OS/2 implements condition variables and they are rather difficult to implement from the primitives provided.

Active Threads provides high-performance multi-object synchronization such as conjunctive acquisition of locks, including reader/writer locks. Neither Solaris, POSIX, nor Java Threads supports this functionality. The efficient deadlock-free implementation of atomic multi-object synchronization is rather tricky [43], yet multi-object synchronization is critical for instance for object-oriented libraries and systems.

Not even user-level thread systems are free of dependencies on the underlying operating systems. For instance, Solaris Threads relies on Solaris for several signals specifically designed for time-slicing. The linker is also modified to recognize threaded applications and link them with special ‘thread safe”C libraries instead of the standard ones. OS/2 contains a built-in interdependency between threads and windows and a system-wide limit of 256 threads! NT threads require compiler changes and in general threaded code cannot be compiled by any compiler.

Both NT and Solaris Threads run on several hardware platforms such as SPARC and x86 for Solaris, x86, ALPHA, and HPPA for NT. The implementation details are, in general, not disclosed, and it is hard to determine how much code is reused in different implementations. Java achieves portability by providing the user with a very narrow interface and relying on native thread systems for different implementations. Active Threads are ported by implementing the Portability Interface on the intended platform - usually several hundred lines of code.

7 MICROBENCHMARKS

We have measured the performance of Active Threads on a variety of hardware platforms: different models of SPARC symmetric multiprocessors, Intel Pentium Pro, DEC Alpha AXP, HPPA 1.1 (Table 5).

Operation	UltraSPARC-1, 167 Mhz	Intel Pentium-Pro, 200Mhz	DEC Alpha AXP 250Mhz	HPPA 9000/755, 99Mhz
thread create	1.3	1.4	1.0	2.0
null thread	5.6	4.4	2.9	7.2
context switch	1.7	1.5	1.1	3.0
uncontested mutex	0.4	0.5	0.3	1.0
uncontested sema.	0.4	0.5	0.3	1.0
mutex try	0.2	0.2	0.1	0.3
semaphore try	0.2	0.2	0.1	0.3
mutex ping-pong	6.0	3.4	2.9	7.9
sema. ping-pong	6.0	3.7	2.8	8.5

Table 5: Performance of Active Threads on different platforms, µs

Thread creation overhead as presented in Table 5 includes thread stack allocation. Thread creation overhead with lazy stack allocation is somewhat smaller. For comparison, a null procedure call on the UltraSPARC-1 takes 0.75µs when register window overflow occurs and 0.08µs without window overflow.[1] Thread creation overhead is almost as expensive as null function call with a window overflow and only about an order of magnitude more expensive than a null call that does not cause a window overflow.

Performance of Active Threads relative to vendor thread packages is shown in Figure 19.This comparison is important since "native"thread systems are usually chosen as compilation targets for concurrent object-oriented languages.

The benchmark operations are the following.

1. gcc v2.7.1, compiled with -O2

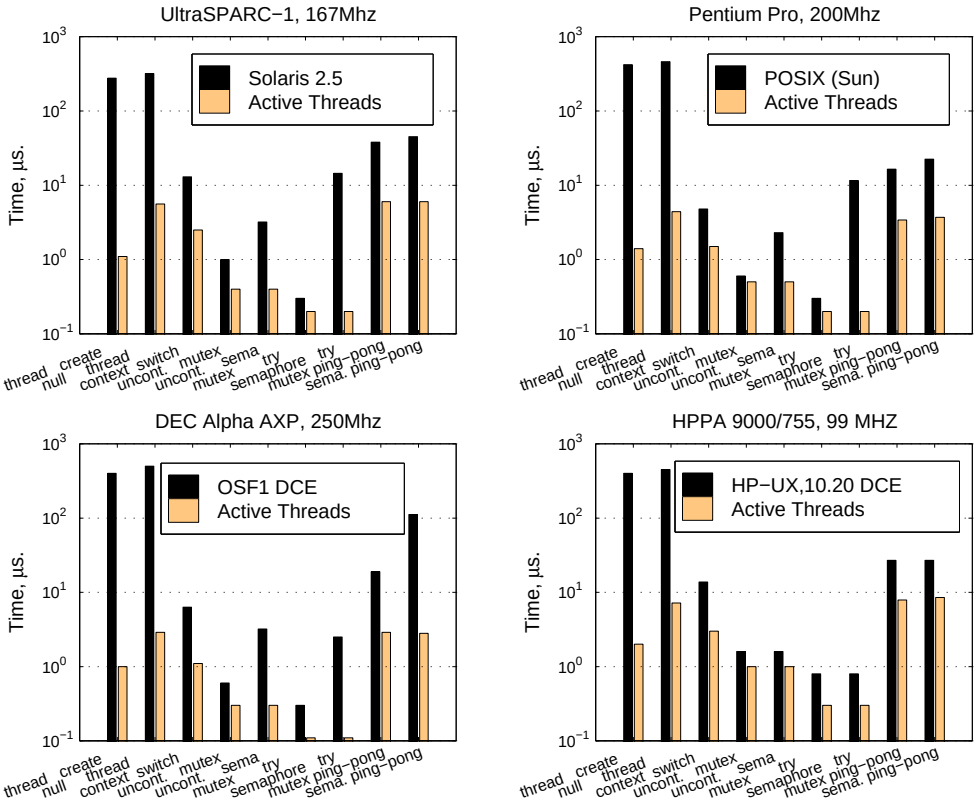

Figure 19: Active Threads vs. proprietary thread systems.

- The *thread create* operation causes a creation of a new thread, including allocation of a thread stack. It does not include context switch time and execution of a newly created thread.
- The *null thread* benchmark measures the entire runtime of thread that performs a null call from creation to termination.
- *Context switch* time is measured by having a thread yield execution to another thread.
- *Uncontested mutex* and *uncontested semaphore* benchmarks time mutex lock and semaphore wait operations in the absence of contention.
- Mutex and semaphore *try* benchmarks measure the overhead of the corresponding non-blocking operations.
- Mutex and semaphore *ping-pong* operations repeatedly synchronize two threads with one another in a manner similar to that used to measure synchronization cost of Solaris Threads in [42]. Ping-pong timings include synchronization of two threads. Per thread synchronization overhead as reported in [42] is exactly one half of the reported numbers.

We have repeated the above measurements on different platforms using the "native"user-level thread packages bundled with proprietary operating systems: Solaris Threads on the SPARC platforms, the Sun Microsystems implementation of POSIX threads on Intel PentiumPro, OSF DCE and HPUX DCE threads on DEC Alpha and HPPA platforms respectively. In all benchmarks, Active Threads substantially outperformed vendor-supplied threads packages (Figure 19).

We also compare Active Threads with Ariadne, a portable light-weight thread package recently developed at Purdue University [35][36]. For the purposes of this comparison, measurements were performed on the same platform as the ones reported in [35], a 4cpu SPARCstation 20. For a reference, performance of Solaris Threads under Solaris 2.5 is also presented. Similar to [35], average times over 1000 operations are reported.

Operation	Solaris Threads	Ariadne	Active Threads
null thread	1715	40	14
thread create	1620	35	3.5
context switch	30	15	4.3
synchronization	43	20	9

Table 6: Comparisons with other systems, microseconds. SPARCstation 20.

The *null thread*[2] operation involves the creation and immediate execution of a null thread, with control returning to the caller. *Thread create* is the same as discussed previously - it is simply the overhead incurred by the parent thread before it can continue. *Synchronization* time is measured by synchronizing two threads repeatedly with each other [36]. It is similar to a semaphore ping-pong operation.

Large creation times for Solaris Threads were observed for default invocation of *thr_create*. By default, the thread stack creation and management are performed by Solaris Threads. This includes setting up "red zones"of unmapped pages following the thread stacks. Solaris Threads perform significantly better if preallocated stacks are provided for thread creation primitives. For instance, the *null thread* and *thread create* operations take 136 and 58 us respectively for preallocated stacks.

Active Threads performs significantly better than Ariadne. Ariadne achieves portability for context switches through the use of the setjmp/longjmp mechanism provided by the C library. This usually requires saving values of all registers. However, depending on the parameter passing convention, only a relatively small number of registers actually needs saving. For instance, the SPARC calling convention prohibits passing parameters in floating point registers. Across a function call, either a caller must save its live floating point registers, or the callee must save the ones it is going to use and restore them before returning. If the blocking operation is implemented as a function call, the compiler must emit code to

2. Solaris and Active Threads timings include creation of new stacks while reported Ariadne numbers are for preallocated stacks.

ensure that all used floating point registers are saved and restored correctly across this call. Thus, the context switch code should not be concerned with saving the 32 floating-point registers since the used registers must be already spilled to memory by the code emitted by the compiler. Similarly, only a portion of global registers needs saving.

Active Threads achieves portability for context switches by retargeting a small but critical portion of the code that may be written in assembly. The actual implementation is based on earlier work by David Keppel [27]. Machine dependent code is usually only a few dozen instructions. Such specialization results in superior performance of Active Threads relative to Ariadne and Solaris Threads.

8 PERFORMANCE STUDIES

This section examines the performance of several Active Threads based applications:

- two kinds of sorts: parallel mergesort and quicksort
- conservative manufacturing simulations
- SPLASH-2 suite applications

Locality issues receive special consideration. For some applications, performance comparisons of several versions based on different runtime systems (Active Threads, POSIX, and Cilk) are performed.

8.1 Sorts

Scheduler Customization

To quantify the benefits of customized schedulers, we have implemented and measured performance of two fairly simple parallel applications: quicksort that switches to bubble sort for leaf nodes and mergesort that switches to insertion sort for leafs. There are obvious similarities between the two applications: both are recursive with recursive invocations being natural units of parallel work. Both implementations employ a fairly fine-grain decomposition - separate threads are created for each recursive subdivision of the input. When the size of input falls below a threshold level, both applications switch to elementary $O(n^2)$ sorts. However, there are also important differences. Quicksort does some processing of the input before splitting it up and passing the two pieces to child threads. Mergesort, on the other hand, evenly divides the input between the child threads and does some processing (merges the lists) after child threads terminate. At each recursive level, mergesort splits the work into two equal parts while quicksort generates unequal work units. Our implementation of quicksort uses an array based repre-

sentation while mergesort deals with the linked list representation. These differences allow us to investigate the influence of affinity annotations in somewhat dif ferent contexts.

The same application sources were compiled and linked with several different thread schedulers:

- FIFO
- FIFO with MCS (memory-conscious scheduling)
- FIFO with lazy thread stack allocation
- FIFO with lazy stack allocation and MCS
- LIFO
- LIFO with MCS
- LIFO with lazy thread stack allocation
- LIFO with lazy stack allocation and MCS

In all cases, threads were organized in a single bundle. In fact, the only difference between the cases was the nature of the bundle.

The FIFO scheduler uses a simple bundle that keeps threads in a queue and satisfies the processors's requests for threads in the FIFO order without regard for data locality. This is similar to the scheduling policies of many modern thread packages. FIFO with MCS uses processor affinity annotations to push threads, as they are created, onto the processor local dispatch queues. The annotations reflect the fact that child and parent threads usually work on the same data and should be scheduled to run on the same processor, if possible. We have also investigated the effects of lazy stack allocations on the two scheduling policies described above.

Finally, we have repeated all experiments, but replaced FIFO-based schedulers with LIFO-based schedulers.

Performance

Figure 20 and Figure 21 present performance measurements for mergesort and quicksort respectively obtained on the Sun Enterprise 5000 with eight 167Mhz UltraSPARC-1 processors. Mergesort sorted a linked list of 100,000 records and quicksort worked on the array of 100,000 records. Both sorts switch to elementary sorts when input size falls below 10.

In the figures, the performance of schedulers with and without MCS is represented by the lines of the same type. The top lines correspond to run times measured for unmodified base schedulers such as LIFO or FIFO and the bottom lines of the same type represent the behavior of the same schedulers with MCS. This allows us to reason about the relative importance of the cache locality issues for different kinds of base schedulers.

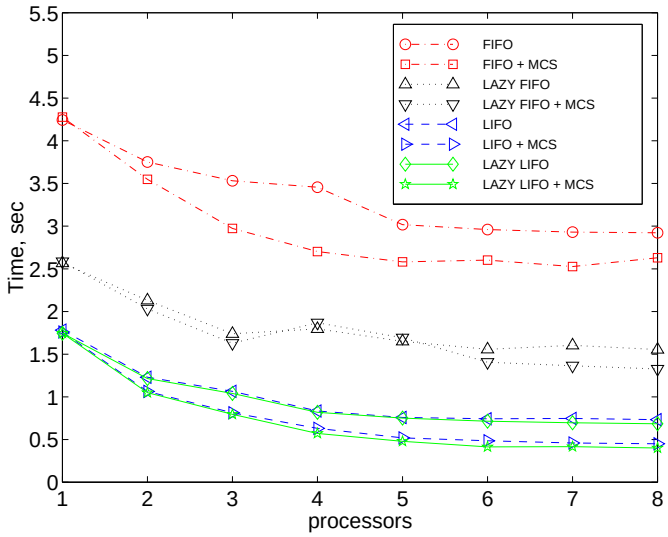

Figure 20: Mergesort performance. 100,000 elements, leaf size 10.

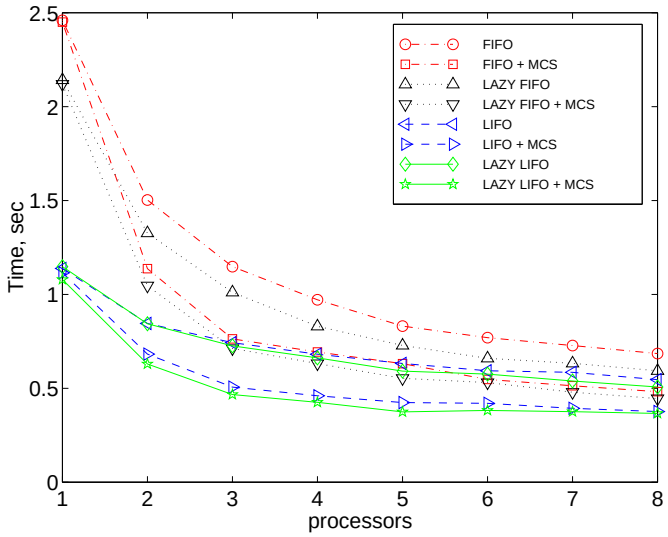

Figure 21: Quicksort performance. 100,000 elements, leaf size 10.

Several lessons can be learned from the figures. First, the details of thread scheduling are indeed important for fine-grained multithreaded applications. For instance, by departing from the standard FIFO policy, we can speed up mergesort by a factor of 6 on the 8cpu E5000. Secondly, FIFO schedulers (and related round-robin schedulers) which are in many

cases the single scheduling policy supported by thread systems, are not suitable for tree structured computations. These computations, for instance, include all branch-and-bound applications. Round-robin scheduling is an important scheduling class and may be unavoidable, for example to ensure that no GUI thread starves indefinitely, however it is hardly appropriate as the only scheduling policy.

The figures suggests that a LIFO scheduler has significant impact on the performance of both mergesort and quicksort. LIFO thread scheduling may be thought of as executing the tree roughly in the depth-first order. This results in a smaller number of runnable threads at each point in time than the FIFO scheduling (FIFO roughly corresponds to a breadth-first execution of a tree). As we will see shortly, fewer threads in the runnable state require less memory for stacks. Reduced stack and memory management overhead explains why the version of mergesort utilizing LIFO thread scheduling performs much better than the original FIFO based implementation.

Adding lazy stack allocation improves performance for similar reasons. It also adds an additional locality benefit. If stacks are allocated at thread startup, stacks from recently terminated threads are recycled. Such stacks are more likely to be already cached by the processor.

Combining various scheduling policies with MCS also had a significant performance impact. In the figures, curves corresponding to scheduling policies different only in the presence or absence of MCS use the same line types. We will investigate the effects of MCS for applications displaying different memory access patterns further in the following sections.

Performance Impact of the Memory Hierarchy Levels

While moving from FIFO to LAZY FIFO and from FIFO to FIFO+MCS we achieve better performance by exploiting different levels of the memory hierarchy. The LAZY FIFO version outperforms our original FIFO based version because of the virtual memory effects. LAZY FIFO is characterized by reduced memory usage due to better thread stack reuse. Such reduction results in significantly fewer page faults and associated OS trap handling overhead. Figure 22 repeats the runtime curves for quicksort using FIFO, LAZY FIFO and FIFO+MCS schedulers. The right part of Figure 22 presents the number of page faults for the three considered scheduling policies.

Better performance of LAZY FIFO relative to FIFO is mostly due to a significantly smaller number of page faults encountered by the LAZY FIFO version. However, better performance of FIFO+MCS scheduler relative to the original FIFO scheduler is due to somewhat more subtle cache effects. As Figure 22 shows, FIFO and FIFO+MCS versions incur roughly the same number of page faults. However, FIFO+MCS version reuses the individual processor's cache significantly better. In the quicksort context, cache effects turn out to be more important than the virtual memory effects and FIFO+MCS version outperforms the LAZY FIFO version in spite of the higher memory usage.

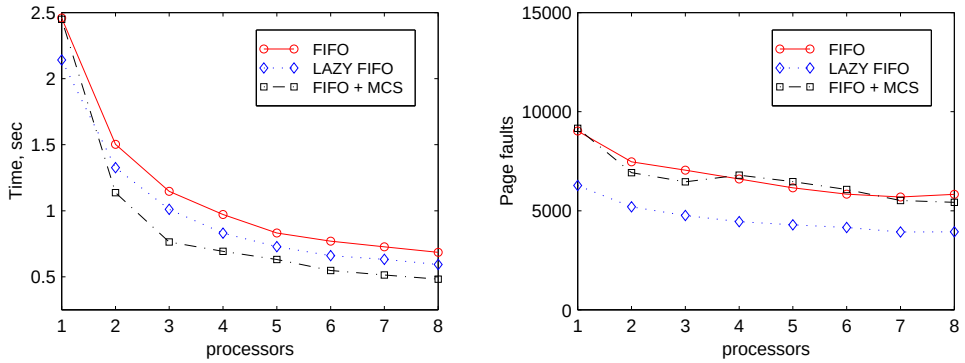

Figure 22: Quicksort: performance effects of the memory hierarchy levels: page faults vs. cache

Memory Usage

The fine-grained multithreading programming style encourages creation of many more runnable threads than physical processors to keep all processors busy. Unless the threads are scheduled carefully, parallel execution may require much more memory than sequential execution. In fact, memory is one of the factors effectively limiting the degree of multithreading. It is not unusual for programs that create many threads to run out of virtual memory, not just physical memory. For such programs, memory usage is no longer just a performance factor - these programs simply cannot execute without additional hardware.

Figure 23 and Figure 24 show the maximum memory usage of mergesort and quicksort respectively under all considered scheduling policies. The space performance correlates well with the time performance considered earlier. A combination of LIFO and MCS that supports lazy stack allocation is not only the fastest, but also requires an order of magnitude less memory necessary for the FIFO version of mergesort and only a third of memory necessary for the FIFO version of quicksort. Better absolute time performance of lazy schedulers is largely explained by its better memory performance relative to schedulers that use eager thread stack allocation. Greater memory consumption increases the absolute overhead of memory management. This is due both to the overhead of maintaining the internal state of the memory allocator and the increased number of page faults.

As expected, lazy thread stack allocation significantly reduces the memory requirements. LIFO scheduling is also beneficial for memory utilization. The FIFO policy, on the other hand, places a high burden on memory resources. FIFO is equivalent to executing the task tree in a roughly breadth-first order. In fact, it is exactly the breadth-first order in a uniprocessor case. The breadth-first execution order is the worst possible policy with respect to memory consumption. In the case of breadth-first execution, the maximum memory requirement is determined by the size of the entire task tree since all created threads are alive at some execution point. The memory requirements for FIFO scheduling decreases somewhat as the number of processors increases. This happens because the breadth-first execu-

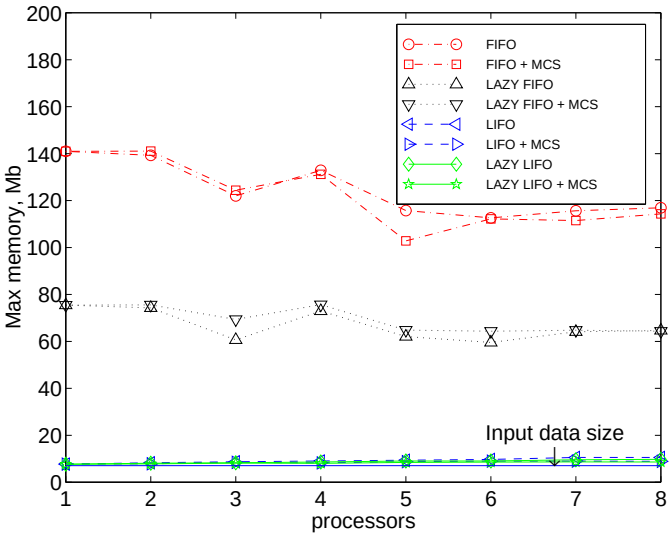

Figure 23: Mergesort: maximum memory requirement. 100,000 elements, leaf size 10.

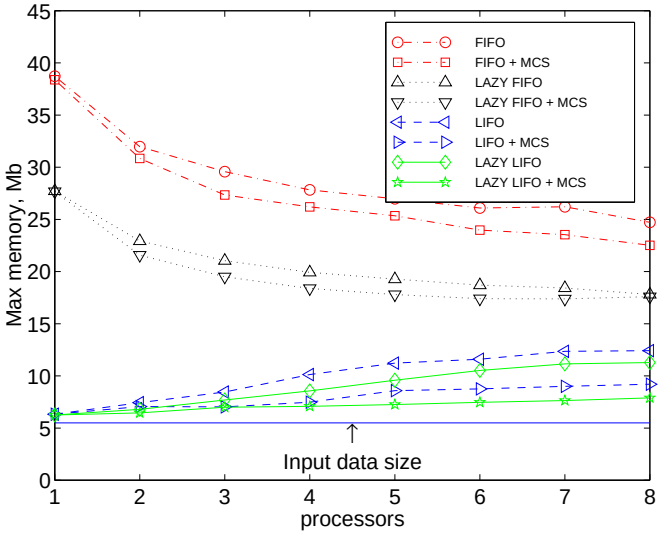

Figure 24: Quicksort: maximum memory requirement. 100,000 elements, leaf size 10.

tion order gets relaxed. The greater the number of processors, the larger the departure from the breadth-first evaluation order. Quicksort subdivides tasks into subtasks of different sizes (the sizes are determined by the pivot element). In the parallel case, tasks are picked up by the processors and processed as they are created. New child tasks are created as parent tasks are processed and this may depart farther from the depth-first order.

A combination of lazy LIFO scheduling with MCS significantly reduces the maximum memory requirement and makes the memory overhead due to multithreading relatively small (about 10% of the input data size for mergesort and 20% of the input size for quicksort). The overhead increases somewhat with the number of processors because it is determined by the number of tasks in the execution tree between the running threads and the root. The greater the number of processors, the greater the number of running threads and consequently the number of tasks on the paths from the running tasks in the execution tree. In the case of a balanced tree, the number of such outstanding tasks is the same order as the number of leafs. Hence, in this case, the maximum memory requirement grows linearly with the number of processors.

Memory-Conscious Scheduling and Memory Access Patterns

Our sorting examples provide a good testbed for the analysis of the MCS impact on applications with different memory access patterns. To illustrate the issue, we consider quicksort from the memory access pattern standpoint. Quicksort switches to simple bubble sort whenever input size falls below a threshold. Bubble sort uses $O(n^2)$ comparisons and exchanges for input of size n. Even if we assume that in the absence of MCS, the first access to an element is always a cache miss, once it is loaded into cache, bubble sort performs $O(n)$ operations on this element. For large leaf sizes n, a penalty of the initial cache miss is amortized over a large number of operations. Thus, by varying the size of leaf nodes, we can change the overall memory access pattern from very irregular - a few operations performed on each element for small leafs - to very regular for large leafs.

The left part of Figure 25 shows how the overall performance of quicksort depends on the leaf size in the presence and absence of memory-conscious scheduling (the base algorithm is LIFO). The right part of Figure 25 displays how the benefits of MCS depend on the regularity of memory access patterns. The size of input was fixed at 2,000,000 records while the leaf size was varied from 5 to 150.

As expected, the benefits of memory-conscious scheduling are the largest for very small leafs since this corresponds to highly irregular memory access patterns. The relative speedup due to MCS falls for large leafs because of the amortization effects of cache misses. As mentioned earlier, once an element is loaded in cache, it is used in $O(n)$ operations. The larger the leaf size n, the less the relative effects of the initial miss. For instance, given the memory access latency of 50 cycles for our Sun Enterprise 5000 and assuming that exactly n operations are performed once an element is loaded into cache, the overhead of a single miss is $\frac{50}{5} = 10$ cycles per operation for leaves of size 5 and only $\frac{50}{100} = 0.5$ cycles per operation for leaves of size 100. More fine-grained threads suffer from locality mismanagement to a greater degree because they simply do not execute long enough before blocking or termination to amortize the cache miss overhead.

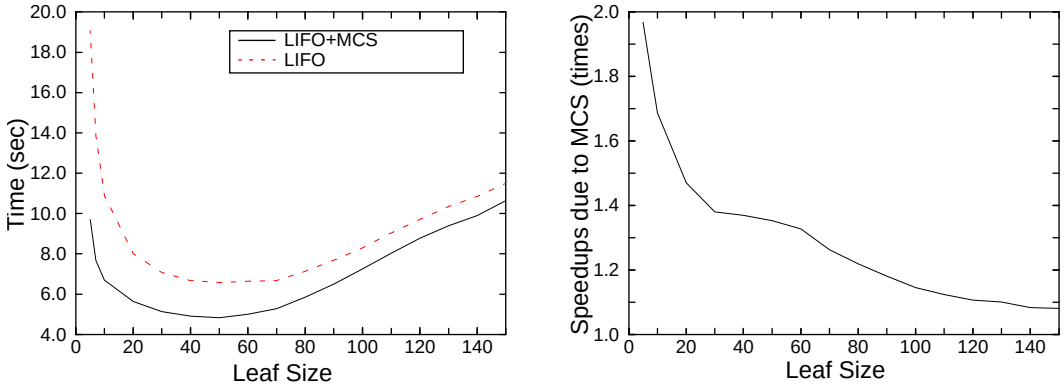

Figure 25: Memory-conscious scheduling and different access patterns. 8cpu Enterprise 5000.

The above argument suggests that the speedups should be greater for larger leaves, which is indeed the case (Figure 26). However, it is a well established fact that serial quicksort performs the best for mid-range leaf sizes [44]. As we see in Figure 25, this is also the case for our simple parallel version. What is especially interesting is that memory-conscious scheduling brings significant improvements even for mid range leaf sizes that yield optimal overall performance.

The quicksort example illustrates the well known fact that speedups are not always the best measure of overall performance.

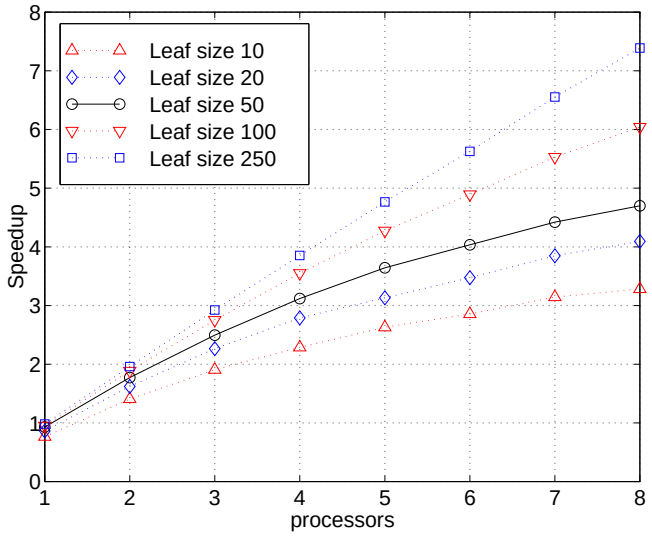

Figure 26: Quicksort: speedup curves for different leaf sizes.

Figure 25 suggests that the highest performance of a parallel version of quicksort is achieved for mid-range leaf sizes: from 40 to 60. However, Figure 26 demonstrates that the highest speedup is observed for larger leaf sizes. The speedups in Figure 26 are computed relative to true serial versions with corresponding leaf sizes in which thread creations were replaced with direct calls to the quicksort routine and thread synchronization was removed. As a result, threaded versions on a single processor have speedups somewhat less than one due to thread creation and synchronization overhead absent in a true serial implementation. As the leaf size increases, thread creation overhead gets amortized over the time necessary for the elementary sorts to sort larger leaves. The speedup curves asymptotically approach a perfect straight line with the increasing leaf size. However, the highest overall performance is achieved for leaf size 50 which corresponds to a solid line curve in Figure 26.

To illustrate scalability of Active Threads applications and the low overhead of thread creation, we present the total number of created threads and runtimes as function of leaf sizes in Figure 27. For instance, our quicksort implementation creates 1,392,931 threads to sort 2,000,000 elements with the leaf size set to 5.

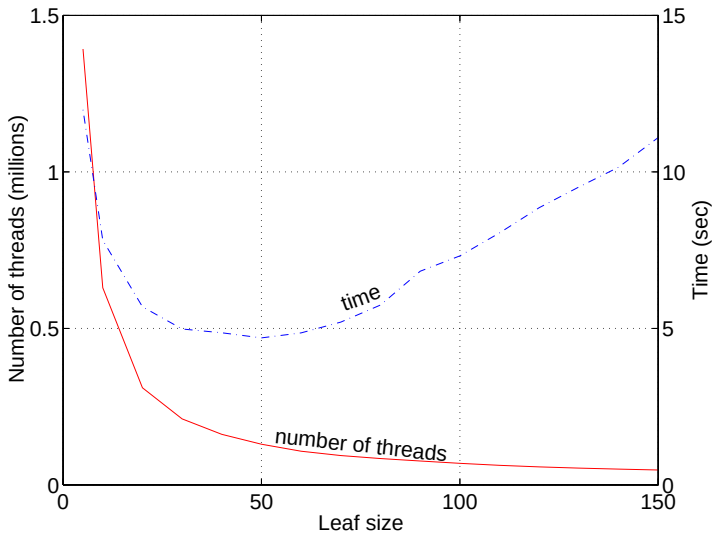

Figure 27: Quicksort: number of threads and performance.

Memory-Conscious Scheduling and Portable Performance

The performance measurements of the previous sections indicate that customized schedulers exploiting memory locality can yield significant benefits. This is especially relevant for applications with irregular memory access patterns. Since we used no specific information about the underlying hardware, we expect that the discussed techniques have a performance impact over a range of platforms with different memory hierarchy organizations.

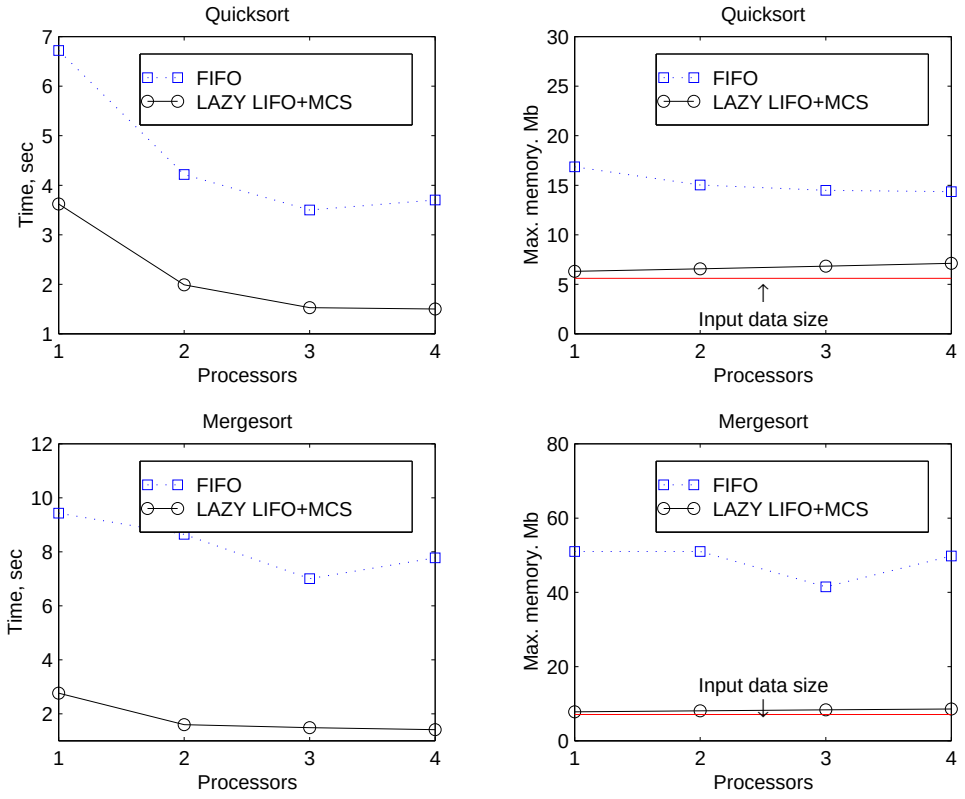

Figure 28: Quicksort (top) and mergesort (bottom) on 4cpu Ross HyperSparc. 100,000 elements, leaf size 10.

To illustrate the portable nature of memory-conscious scheduling in Active Threads, we have repeated the original quicksort and mergesort experiments on a SparcStation 10 with 4 Ross 50Mhz HyperSparc processors. The two considered platforms have quite different memory hierarchy architectures. Each processor of our model of Enterprise 5000 is a 167Mhz UltraSPARC-I with 16KB of direct mapped write-through on-chip cache and 1MB of external cache organized in 64 byte lines. External cache access consumes only three cycles and returns 16 bytes of data per cycle. Memory access latency is 50 cycles if the word is not cached by another processor and 80 cycles otherwise. Multiple accesses to external cache are pipelined [61][62]. In contrast, each Ross HyperSparc processor has only 256K of L1 cache with 32 byte lines.

Despite the differences, our scheduling techniques achieved significant performance improvements for a 4 cpu SparcStation 10. For instance, a version of mergesort with a lazy LIFO+MCS scheduler runs about an order of magnitude faster and needs about an order of magnitude less memory than the one with a simple FIFO scheduler (Figure 28.) Timings were measured for exactly the same applications and data sets as the ones used on the Enterprise 5000, with no tuning for a different memory hierarchy.

8.2 Conservative Manufacturing Simulations[3]

The objective of this study is to evaluate how parallel and distributed simulations techniques can be applied in a virtual factory simulation [25]. The simulations include the modeling of manufacturing and business processes, and communications network in a production plant. Such a simulation environment will allow one to model and analyze the effects of different system configurations and contr ol policies on actual system performance. The initial focus is on the Singapore electronics industry.

In this section, we describe performance results for a parallel discrete event simulation (PDES) of a simplified, but generic version of a manufacturing process, such as that found in the semiconductor wafer manufacturing. In a PDES, a physical process is modeled by a *logical process (LP)*, and events in the physical system are simulated by communications between LPs using timestamped messages. An LP may not always receive messages with increasing timestamps, but, in order to correctly simulate the physical system, it must process the messages in the global timestemp order. In a conservative approach, an LP can make progress only when the causality is preserved. In an optimistic approach, an LP is allowed to proceed with the simulation as far forward as possible. However, if the violation of causality is detected, an LP has to roll back in the simulation time.

We now turn to performance of the conservative simulations implemented based on three different parallel libraries:

- A Sun implementation of the POSIX Thread standard [23].
- Cilk (MIT), a non-blocking atomic thread library with provably good scheduling and load balancing mechanism [5].
- Active Threads

3. Applications considered in this section have been designed and implemented at the Gintic Institute of Manufacturing Technology and the School of Applied Science in Nanyang Technological University, Singapore. The data has been generously provided by Chu-Cheow Lim and Yoke-Hean Low of Gintic and used with their permission.

.

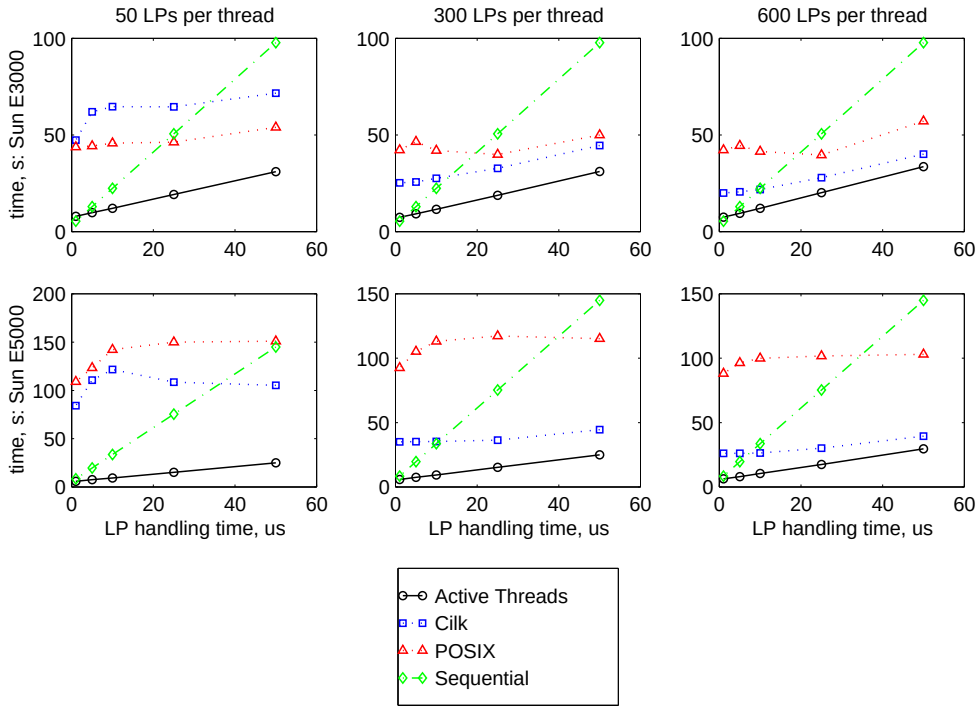

Figure 29: Performance of conservative event simulations on 4 CPU Sun Enterprise 3000 (top row) and 8 CPU Sun Enterprise 5000 (bottom row).

A detailed account of advantages and disadvantages of the considered systems for conservative discrete simulations and a full description of the simulation algorithm and data structures are presented in [29]. Both Active Threads and POSIX were quite well suited for the simulations (and, in fact, shared most of the implementation code).

In all experiments, each thread simulates a certain number of LPs. The timings for the different numbers of LPs per thread (50, 300, and 600) are given in Figure 29. Each logical processor handles incoming events and the figure reflects the running time as a function of the event handling interval. The top row corresponds to the timings obtained on the Sun Enterprise 3000 with four 250Mhz UltraSPARC-2 cpus. The bottom row represents the measurements obtained on the Sun Enterprise 5000 with eight 167Mhz UltraSPARC-1 cpus.

Four different implementations of the simulations have been built: a true sequential implementation free of any thread and synchronization calls and parallel implementations based on POSIX threads, Active Threads, and Cilk threads. As Figure 29 shows, the Active Threads implementation substantially outperforms all others and displays good speedups

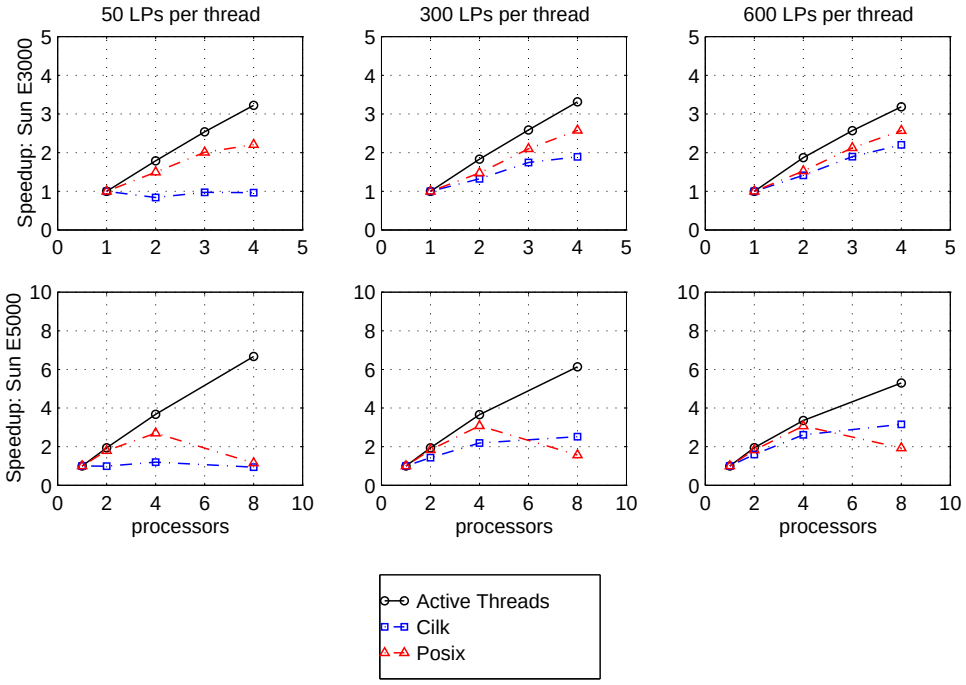

Figure 30: Scalability of different parallel runtime systems.

even for the shortest event handling times. The difference between Active Threads and the other system is particularly evident for more fine-grained cases (50 LPs per thread, short event handling interval). It is instructive that Active Threads performs significantly better than Cilk, a celebrated runtime from MIT with non-blocking threadlets and explicit continuations.

No attempt to improve Active Threads performance by using locality scheduling has been done since even the base Active Threads implementation displays substantially higher performance that the other parallel runtime systems [29].

Figure 30 demonstrates how different implementations scale with increasing the number of processors. The speedups in this figure are computed with respect to the 1 processor performance (rather than a true sequential implementation), since we are mostly interested in relative scalability of different implementations

.

Active Threads display fairly good scalability on both considered architectures for all thread granularities. Other systems display speedups only for coarse-grain threads. Active Threads speedups drop somewhat on the 8cpu Sun Enterprise 5000 for coarse-grain threads because load balancing is done at coarser granularities and with only few threads available, processor idle time increases (threads may block to satisfy the simulation causality requirement). However, in general, the speedup curves for Active Threads are much less sensitive to variations in thread granularity.

This comparison study used a simple manufacturing process model. Various aspects of a virtual factory model, including business processes, manufacturing and communications network are currently being integrated into the simulations.

8.3 The Splash-2 Suite

To test the robustness of Active Threads and to demonstrate flexibility, we have ported the SPLASH-2 (Stanford ParalleL Applications for SHared memory) applications suite to Active Threads. SPLASH-2 [45][67] applications are scientific and engineering programs representing various fields: astronomy, oceanography, computer graphics, numerical analysis, etc. The SPLASH-2 suite was designed to provide parallel programs for the evaluation of architectural ideas and trade-offs. Detailed simulation studies of the SPLASH-2 programs on small and medium size symmetric multiprocessors are represented in [67]. A. Tucker [59] performed experimental analysis of the older version of the SPLASH suite on a four processor SGI PowerStation.

Barnes	uses Barnes-Hut hierarchical N-body method to simulate the interaction of a system of bodies in three dimensions over a number of time steps.
FMM	uses the adaptive Fast Multipole Method to simulate the interaction in two dimensions
Ocean	studies large-scale ocean movements based on eddy and boundary currents. Two implementations are provided (1) non-contiguous partition allocation and (20) contiguous partition allocation. More information about the differences in the implementations could be found in [67].
Radiosity	computes the distribution of light in a rendered scene using the iterative hierarchical diffuse radiosity method.
Raytrace	uses ray tracing to render a three dimensional scene.
Volrend	renders a three-dimensional volume using a ray casting technique.
Water-Nsquared	evaluates forces and potentials over time in a system of water molecules using an $O(n^2)$ algorithm.
Water-sp	solves the same problem using a more efficient $O(n)$ algorithm by imposing a uniform 3-D grid of cells on the problem domain.

Table 7: SPLASH-2 applications

The SPLASH-2 suite consists of a mixture of complete applications and short computational kernels. SPLASH-2 currently includes eight complete applications. Short descriptions of these applications are given in Table 7.

The SPLASH applications use PARMACS, a collection of macros designed at the Argonne National Lab to implement parallelism [31][19]. PARMACS is a public domain interface that deals with basic parallel operations: starting parallel execution, synchronization, message passing, safe memory management, etc. A version of PARMACS based on Active Threads was implemented. This automatically achieves portability of programs using PARMACS (including SPLASH) across all platforms supported by Active Threads, which extends portability of the SPLASH suite. The Stanford SPLASH distribution comes with PARMACS macros only for Encore Multimax, SGI 4D/240, and Alliant FX/8.

Figure 31 shows speedups displayed by the SPLASH-2 applications on the 8 cpu Sun E5000. All measurements were performed using the standard input sets distributed with the SPLASH application suite.

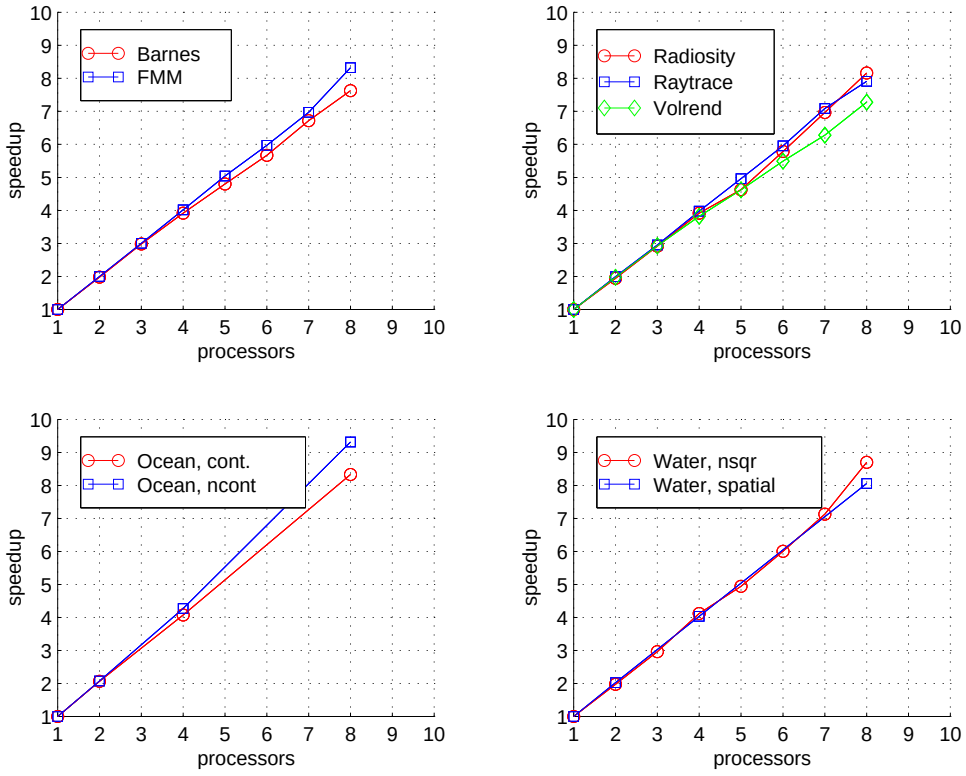

Figure 31: Performance of SPLASH-2 applications on 8 cpu Enterprise 5000.

All applications in the suite display fairly good speedups. This correlates well with the simulation results computed for a perfect memory system [67]. Previous simulation studies showed that the most important working sets should fit in large modern second level caches. In particular, simulations showed that for a 32 node machine all working sets fit inside 1Mb caches with most fitting in 256K caches [67]. Moreover, the same study suggests that the size of the largest working set is proportional to $\frac{DS}{P}$ for most SPLASH applications, where DS is the data set size and P is the number of processors. The exceptions are Radiosity, Raytrace, and Water_Nsquared whose larger working sets do not directly depend on the number of processors. Hence, we can expect that all working sets fit in the 1Mb external caches of the eight processor Enterprise 5000.

With regard to the lightweight thread model, SPLASH applications are not very challenging. All applications are coarse-grained with the number of threads matching the number of processors. Such design favors spinning synchronization objects since no benefit could be gained from thread blocking (in the absence of multiprogramming.) Some SPLASH applications are already tuned to improve data locality. Most use sophisticated data structures for load balancing. While this makes such applications less sensitive to the implementation of the underlying thread and synchronization mechanism, it duplicates functionality already provided by the thread runtime systems.

Speedups displayed in Table 31 leave little room for improvement. In the previous study by A. Tucker, traces from an older version of SPLASH were collected on a four processors SGI PowerStation 4D/340 with 64K L1 cache and 256K L2 cache [59]. The study showed that even if all caches are flushed at the end of each dispatch interval, potential performance loss is only 10-20% for most applications. Since Sun Enterprise 5000 has a much bigger L2 cache, even potential cache losses should be quite small. In short, SPLASH applications are too coarse-grained and too tuned for any further significant performance improvements due to memory-conscious thread scheduling.

9 CONCLUSIONS

Active Threads is a flexible general-purpose threads package that supports fine-grained parallel programming across a variety of hardware platforms. The system was designed to promote compositional software development while preserving high efficiency . Active Threads' support for flexible extensible scheduling allows software modules to be used together with the associated scheduling policies. Negative locality effects whose relative importance increases for fine-grained thr eads are addressed through custom schedulers that can exploit the memory hierarchy.

Active Threads can be used directly by the application and library developers or as a virtual machine for compilers for parallel languages. For instance, Active Threads is a compilation target for Sather[4], a parallel object-oriented language under development at the International Computer Science Institute. Active Threads are also being used as a base for a threaded distributed extension of C++ that supports thread migration [22].

The Active Threads system is fully implemented an runs on several hardware platforms including SPARC, Intel 386 and higher, DEC Alpha AXP and HPPA. Active Threads substantially outperforms vendor-supplied thread systems on all these platforms.

Ongoing work seeks to exploit hardware performance counters that become more common on modern architectures [54] to dynamically guide thread placement. This could be achieved by extending the library of Active Threads schedulers. No modifications to the system itself are necessary. Such functionality is critical for highly irregular applications for which thread placement annotations are difficult or impossible.

4. Currently distributed Sather 1.2 (November 1997) is based on Active Threads and can be downloaded from http://www.icsi.berkeley.edu/~sather

Appendix: Active Threads API

Basic Types

at_thread_t

Active Threads thread type

at_bundle_t

Active Threads bundle type

at_mutex_t

Blocking mutual exclusion lock

at_rw_t

Multiple readers, single writer lock

at_sema_t

Blocking semaphore

at_barrier_t

Blocking barrier

at_cond_t

Condition variable

at_spinlock_t

Spinning mutual exclusion lock (the current implementation first spins on a local copy to minimize bus traffic)

at_hybridlock_t

A hybrid implementation of a mutual exclusion lock. This is essentially a two-phase mutual exclusion lock that combines spinning with blocking. The semantics is the same as at_mutex_t, but the calling thread may spin for a while before blocking. The current implementation uses an exponential back-off policy for the spinning interval.

at_userf_x_t

where x is currently between 0 and 6. A type for a user function used in thread creation. A function takes x arguments of type at_word_t.

at_word_t

A type that maps to a word size entity of the underlying architecture. There is no requirement for the number of bits.

Scheduler types

```
typedef struct at_scheduler {
void (*thread_created)(at_bundle_t *b, at_thread_t *t);
void (*thread_terminated)(at_bundle_t *b, at_thread_t *t);
  void (*thread_started)(at_bundle_t *b, at_thread_t *t);
  void (*thread_blocked)(at_bundle_t *b, at_thread_t *t);
  void (*thread_unblocked)(at_bundle_t *b, at_thread_t *t);
  void (*bundle_created)(at_bundle_t *parent, at_bundle_t *b);
  void (*bundle_terminated)(at_bundle_t *parent, at_bundle_t *b);
  void (*processor_idle)(at_bundle_t *b, int proc);
} at_scheduler_t;
```

This is the only type that a user-supplied scheduler must implement to extend the Active Thread scheduling library. Any scheduler implementation must provide event handlers with the above interfaces for the eight events vectored by the Active Threads runtime: thread created, thread terminated, thread started, thread blocked, thread unblocked, bundle created, bundle terminated and processor idle. Active Threads imposes no restrictions on internals of scheduler data structures.

Basic Thread Operations

at_thread_t *at_create_x(at_bundle_t *b, int affinity, at_userf_x_t *func, at_word_t arg0,...)

Create a new thread of control that will execute a user supplied function *func* with the supplied arguments. The current implementation supports up to 6 arguments. The thread is added to a specified bundle *b*. If *b* supports locality-based scheduling, *affinity* may be used as a virtual processor affinity annotation for a thread. All schedulers that implement some form of locality-based scheduling must accept a full range of virtual processors. AT_UNBOUND can be used if the bundle does not support affinity scheduling, or if the thread does not intend to take advantage of it.

void at_yield()

yield execution to another thread.

void at_exit()

terminate the calling thread

at_thread_t *at_self()

return a pointer to the thread structure of a calling thread

void at_setlocal(void* addr)

set the local memory base address for the currently running thread to a specified value

void *at_getlocal()

get the local memory base address for the currently running thread

void at_stop()

stop execution of any new threads until a call to at_continue(). Threads in progress are not affected until they block or terminate.

void at_continue()

resume thread execution

int at_get_affinity()

return virtual processor affinity of a calling thread.

void at_set_affinity(int vproc)

Changes the affinity of a calling thread to a specified virtual processor. This is conceptually equivalent to thread blocking and resuming on a physical processor to which a specified virtual processor is mapped.

void at_create_local(at_thread_t *t)

creates a new local storage of the size specified at Active thread initialization for a thread *t*. Can be used, for instance, to implementing a lazy allocation policy.

void at_destroy_local(at_thread_t *t)

returns thread's local storage to a pool maintained by Active Threads.

void at_create_stack(at_thread_t *t)

creates a new thread stack for a specified thread.

at_destroy_stack(at_thread_t *t)

returns a supplied thread's stack to a pool of stacks maintained by the Active Threads runtime.

Bundle Operations

at_bundle_t *at_bundle_create(at_bundlle_t *parent, int type)

create a new bundle of a specified type as a child of a *parent* bundle.

void *at_bundle_destroy(at_bundle_t *b)

destroy the bundle which is no longer needed. No threads must be attached to the bundle.

at_bundle_t* at_get_focus()

obtain a bundle that has a current execution focus. A bundle with a focus obtains all events that the Active Threads runtime vectors to thread schedulers. It may handle them or pass them up or down the bundle activation tree for handling.

void at_set_focus(at_bundle_t *b)

set execution focus to a supplied bundle *b*.

Synchronization Objects

Blocking Mutual Exclusion Locks

at_mutex_t* at_mutex_create()

create a new mutual exclusion lock

void at_mutex_init(at_mutex_t *m)

initialize (possibly statically allocated) mutex. Mutex is initialized to the unlocked state

void at_mutex_destroy(at_mutex_t *mutex)

the destroy the mutex which is no longer needed

void at_mutex_lock(at_mutex_t *mutex)

lock the mutex pointer to by *mutex*. If the mutex is already locked, the calling thread blocks until the mutex becomes available.

void at_mutex_unlock(at_mutex_t *mutex)

unlock the mutex pointed to by *mutex.*

int at_mutex_trylock(at_mutex_t *mutex)

attempt to lock the mutex. If successful, locks the mutex and returns 1, otherwise returns 0.

Readers/Writer Locks

at_rw_t* at_rw_create()

create a multiple reader, single writer lock (in the unlocked state).

void at_rw_init(at_rw_t *rw)

initialize (possibly statically allocated) readers/writer lock. The lock is initialized to the unlocked state.

void at_rw_destroy(at_rw_t *rw)

destroy the readers/writer lock

void at_rw_rd_lock(at_rw_t *rw)

Acquire a read lock on the readers/writer lock. If *rw* is already locked for writing, the calling thread blocks until the writer lock is released. Many threads can acquire the reader lock of *rw* at the same time.

int at_rw_rd_trylock(at_rw_t *rw)

Attempt to acquire a read lock on *rw.* Returns 1 if the lock is acquired, 0 otherwise.

void at_rw_wr_lock(at_rw_t *rw)

Acquire a write lock on the readers/writer lock. If *rw* is already locked for either reading or writing, the calling thread blocks until all readers or the writer release the lock. Only a single thread can hold a write lock of *rw* at any time.

int at_rw_wr_trylock(at_rw_t *rw)

Attempt to acquire a write lock on *rw.* Returns 1 if the lock is acquired, 0 otherwise.

void at_rw_rd_unlock(at_rw_t *rw)

unlock the read lock of the readers/writer lock pointed to by *rw.*

void at_rw_wr_unlock(at_rw_t *rw)

unlock the write lock of the readers/writer lock pointed to by *rw.*

Blocking Semaphores

at_sema_t *at_sema_create(int count)

create a counting semaphore and set it to a specified value. *count* must be non-negative.

void at_sema_init(at_sema_t *s, int count)

initialize (possibly statically allocated) semaphore to 'count'

void at_sema_destroy(at_sema_t *sema)

destroy a counting semaphore

void at_sema_wait(at_sema_t *sema)

a calling thread may proceed only if the value of the semaphore is currently greater than 0. If the semaphore value is positive, it is decremented and the calling thread continues. Otherwise, the calling thread blocks until the semaphore counter becomes positive.

int at_sema_trywait(at_sema_t *sema)

a nonblocking version of the previous call. If the semaphore counter is positive, its semantics is equivalent to that of *at_sema_wait*, but it also returns 1. Otherwise, it returns 0 and does not change the semaphore counter.

void at_sema_signal(at_sema_t *sema)

increment the count of a semaphore. If prior to the call, the value of *sema* was 0, and there were threads blocked on the semaphore, one of them is unblocked and allowed to return from its call to *at_sema_wait()*.

Blocking Barrier

at_barrier_t *at_barrier_create(int size)

create a barrier object that becomes "open" when *size* threads try to enter it. As long as the number of such threads is below *size*, the threads are all blocked on the barrier.

void at_barrier_init(at_barrier_t *barrier, int size)

initialize a (possibly statically allocated) barrier to *size*.

void at_barrier_destroy(at_barrier_t *barrier)

destroy a barrier object.

void at_barrier_enter(at_barrier_t *barrier)

If the number of threads that have reached a barrier (including the calling thread) is 'size' (specified during barrier initialization), all thr eads sleeping on a barrier are unblocked. Otherwise, the calling thread blocks on the barrier.

Condition Variables

Condition variables enable threads to block until an arbitrary condition is satisfied. The condition must always be tested under the pr otection of a mutex. When the condition is false, the thread blocks on the condition variable by calling at_cond_wait() and mutex is released by for the thread by the Active Threads runtime. Blocking on the condition variable and releasing the mutex is atomic. Any thread that changes the condition can signal the condition variable the change by calling at_cond_signal(or at_cond_broadcast()

at_cond_t *at_cond_create()

create a new condition variable

void at_cond_init(at_cond_t *c)

initialize (possibly statically allocated) condition variable

void at_cond_destory(at_cond_t *c)

destroy a condition variable

void at_cond_wait(at_cond_t *c, at_mutex_t *mx)

atomically releases the mutex pointed to by "mx" and causes the calling thread to block on the condition variable pointe to by "c". The blocked thread may be subsequently awaken by at_cond_signal() or at_cond_broadcast(). Any change of the associated condition must be reevaluated after a signal unblock a thread.

void at_cond_signal(at_cond_t *c)

unblocks one thread that is blocked on the condition variable pointed to by "c"

void at_cond_broadcast(at_cond_t *c);

unlcoks all threads that are blocked on the condition variable pointed to by "c"

Spinning Mutual Exclusion Locks

AT_SPINLOCK_DEC(s)

declare a spinlock. Spinlocks don't need to be explicitly created or deleted, but they do need to be explicitly initialized before use. The special type *at_spinlock_t* may be used for declarations or typedefs, but BR_SPINLOCK_DEC is preferred when possible.

AT_SPINLOCK_INIT(s)

initialize the spinlock

AT_SPINLOCK_LOCK(x)

lock the spinlock. This may trigger busy-waiting if the spinlock is already locked by another thread. The current implementation first waits on a local cached copy of a spinlock to minimize the bus traffic

AT_SPINLOCK_UNLOCK(x)

unlock the spinlock. If prior to the call there were threads busy waiting on the spinlock, a single thread is allowed to acquire the spinlock and return from a call to AT_SPINLOCK_LOCK()

AT_SPINLOCK_TRY(x)

a non-blocking version. If successful, lock the spinlock and returns 1, otherwise returns 0.

Hybrid Implementation of Blocking Mutual Exclusion Locks

The semantics of hybridlocks is equivalent to that of mutual exclusion locks (but different from that of spinlocks!). Hybridlock perform some busy-waiting if the lock is already locked in an attempt to avoid a context switch. If the lock remains locked, a calling thread eventually blocks. The current implementation uses an exponential back-off waiting policy for the spinning phase.

AT_HYBRIDLOCK_DEC(s)
AT_HYBRIDLOCK_INIT(s)
AT_HYBRIDLOCK_LOCK(x)
AT_HYBRIDLOCK_UNLOCK(x)
AT_HYBRIDLOCK_TRY(x)

Miscellaneous

void at_init(unsigned int concurrency, unsigned int stack_size, unsigned int local_size)

initialize the Active Threads package. Use specified concurr ency level, stack size and local storage size. Concurrency is between 1 and the number of physical processors.

void at_do_when_idle(void (*func)())

register a function to be called when a processor is out of work and there are no work thread to run. This could be used, for example, to periodically service the network or perform incremental garbage collection.

int at_ncpus()

returns the number of physical processors

int at_vproc()

returns the current virtual processor number, or -1 if the thread is unbound. Could be thought of as an alias for at_get_affinity(). The virtual pr ocessor number has no relation to the physical processor number. When a thread is created, it can be assigned to a virtual processor. If the bundle supports memory-conscious scheduling (default does not), it will try to run threads bound to the same virtual cpu to run on the same physical cpu. Virtual processor numbers can be arbitrary large.

int at_cpu()

returns the physical processor on which the calling thread is executing

References

[1] T. E. Anderson, B. N. Berhad, E. D. Lazowska, H. M. Levy, **Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.** *ACM Trans. Comput. Systems* 10(1), Feb. 1992

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. M. Levy, **Thread Management for Shared Memory Multiprocessors**. To appear, Handbook for Computer Science. Also available from http://http.cs.Berkeley.EDU/~tea.

[3] B. N. Bershad. **The Presto User Manual.** October 1991. Available from http://www.cs.washington.edu/research/compiler/papers.d/presto.html

[4] G. E. Blelloch, S. Chattejee, J. C. Hardwick, J. Sipelstein, and M. Zagha, **Implementation of a Portable Nested Data-Parallel Language.** *Journal of Parallel and Distributed Computing*, 21(1):4-14, April 1994.

[5] R. D. Blumofe, C. E. Leiserson, **Scheduling Multithreaded Computations by Work Stealing.** *35th Annual IEEE Conference on Foundations of Computer Science (FOCS94)*, Santa Fe, New Mexico, November 1994.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and J. Zhou. **Cilk: An Efficient Multithreaded Runtime System**, *Journal of Parallel and Distributed Computing*, Vol. 37. No 1, August 1996. pp 55-69.

[7] K. M. Chandy, C. Kesselman, **Compositional C++: Compositional Parallel Programming.** I*n Proc. 5th International Workshop on Languages and Compilers for Parallel Computing*, pp124-144, New Haven, CT, August 1992.

[8] D. E. Culler, **Managing Parallelism and Resources in Scientific Dataflow Programs.** *Technical Report 446*, MIT Lab for Comp. Sci. March 1990.

[9] D. E. Culler, A. Sah, K. E. Schauser. **Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine.** *Fourth Internatonal Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, Ca, April 1991.

[10] H. Custer, **Inside Windows NT**, pp. 94-100. Microsoft Press, 1993

[11] E. W. Dijkstra, **Cooperating Sequential Processes.** *Programming Languages*, pp. 43-112. Academic Press, 1968.

[12] R. Draves, E. Cooper, **C Threads**. *Technical Report. CMU-CS-88-154.* School of Computer Science, Carnegie Mellon University, June 1988.

[13] S. C. Goldstein, **The Implementation of a Threaded Abstract Machine.** *Technical Report?*, University of California at Berkeley. May 1994.

[14] S. C. Goldstein, D. E. Culler, K. E. Schauser, **Lazy Threads, Stacklets, and Synchronizers: Enabling primitives for compiling parallel languages.** *Technical Report*, University of California at Berkeley, 1995.

[15] S. C. Goldstein, K. E. Schauser, D. E. Culler, **Lazy Threads: Implementing a Fast Parallel Call.** *Journal of Parallel and Distributed Computing*, Vol. 37. No 1, August 1996. pp. 5-20.

[16] B. Gomes, D. P. Stotamire, B. Weissman, and H. Klawitter, **Sather 1.1 Language Essentials**. *International Computer Science Institute.* Available at http://www.icsi.berkeley.edu/~sather/Documentation/LanguageDescription/contents.html.

[17] R. H. Halstead. **Mutilisp: A Language for Concurrent Symbolic Computation.** *ACM Transactions on Programming Languages and Systems,* 7(4):501-538, October 1985.

[18] B. Hansen, **The Programming Language Concurrent Pascal**. *IEEE Transactions on Software Engineering* 1(2):199-207 (Jun 1975)

[19] R. Hempel, **The ANL/GMD Macros (PARMACS) in FORTRAN for Portable Parallel Programming using the Message Passing Programming Model. User's Guide and Reference Manual. Version 5.1.** Gesellschaft fur Mathematik and Datenverarbeitung mbH. November 1991.

[20] J. M. D. Hill. **Installation and User Guide for the Oxford BSP toolset (v1.1) implementation of BSPlib.** *Oxford Parallel Computing Laboratory*, Oxford University. June 1997.

[21] High Performance Fortran Forum. **High Performance Fortran Language Specification** , May 1993.

[22] M. Holtkamp, **Thread Migration with Active Threads.** International Computer Science Institute, Technical Report 1997, TR97-038

[23] The Institute of Electrical and Electronics Engineers. **Portable Operating System Interface (POSIX) - Part 1: Amendment 2: Threads Extensions [C Language].** *POSIX P1003.4a/D7. April, 1993*

[24] **Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference.** Intel Corporation, Order Number 243191. January 1997

[25] S. Jain. **Virtual Factory Framework: A Key Enabler for Agile Manufacturing.** *1995 INRIA/IEEE Symposium on Emerging Technologies and Factory Animation.* Paris, October 1995. Vol. 1, pp. 247-258, IEEE Computer Society Press, Los Alamitos, CA.

[26] D. Keppel, **Register Windows and User-Space Threads on the SPARC**. *Department of Computer Science and Engineering, University of Washington.* Technical Report UWCSE 91-08-01, August 1991.

[27] D. Keppel, **Tools and Techniques for Building Fast Portable Threads Packages**. *University of Washington,* Technical Report UWCSE 93-05-06.

[28] B. Lewis, D. J. Berg, **Threads Primer. A Guide to Multithreaded Programming.** p. 1. Sun Soft Press 1996.

[29] C. C. Lim, Y. H. Low, W. Cai, W. Hsu, S. Y. Huang. **An Empirical Comparison of Runtime Systems for Conservative Parallel Simulation.** *Submitted to the 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP),* in conjunction with the 12th International Parallel Processing Symposium (IPPS/ SPDP 1998), March 1998.

[30] D. K. Lowenthal, V. W. Freeh, G. R. Andrews. **Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing**, *Journal of Parallel and Distributed Computing*, Vol. 37. No 1, August 1996. pp 41-54

[31] E. Lusk, et al. **Portable Programs for Parallel Processors.** Holt, Rinehart and Winston, Inc; New York, 1987.

[32] E. P. Markatos, T. J. LeBlank, **Locality-Based Scheduling for Shared-Memory Multiprocessors.** Institute of Computer Science, Creete Greece, FORTH-ICS/ TR-094. Also appears in Zomaya (Ed.) Current and Future Threands in Parallel and Distributed Computing. World Sceintifc Publishing, 1994.

[33] E. P. Markatos, **How Architecture Evolution Influences the Scheduling Discipline used in Shared-Memory Multiprocessors.** *Parallel Computing 1993.*

[34] B. D. Marsh, T. J. LeBlanc, M. L. Scott, E. P. Markatos, **First-Class User-Level Threads.** 1*3th ACM Symposium on Operating Systems Principles,* October 1991.

[35] E. Mascarenhas, V. Rego, **Migrant Threads on Process Farms: Parallel Programming with Ariadne.** *Technical Report TR95-081*. Department of Computer Sceinces, Purdue University. Decmber 1995.

[36] E. Mascarenhas, V. Rego, **Ariadne: Architecure of a Portable Threads System Supporting Thread Migration.** *Software - Practice and Experience,* VOL. 26(3), 327-356 (March 1996)

[37] H. Massalin. C. Pu, **Threads and Input/Output in the Synthesis Kernel.** *12th ACM Symposium on Oerating Systems Principles*, December 1989, pp. 191-201.

[38] E. Mohr, D. A. Kranz, R. H. Halstead, **Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs.** *IEEE Transactions on Parallel and Distributed Systems*, 1990.

[39] D. Nussbaum, **Run-Time Thread Management for Large-Scale Distributed-Memory Multiprocessors.** Ph.D. Thesis. MIT, 1993

[40] S. Oaks, H. Wong, **Java Threads**. O'Reilly, 1997.

[41] **Pentium Pro Family Developer's Manual. Volume 3: Operating System Writer's Guide.** Intel Corporation, Order Number 242692. December 1995.

[42] M. L. Powell, S. R. Kleinman, S. Barton, D. Shah, D. Stein, M. Weeks. **SunOS 5.0 Multithreaded Architecture.** *A White Paper*. Sun Microsystems, 1991.

[43] J. W. Quittek, B. Weissman, **Efficient Extensible Synchronization in Sather .** To appear in *The 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97).* December 1997.

[44] R. Sedgewick, **Algorithms**. Addison-Wesley, 1988.

[45] J. P. Singh, W. D. Weber, A. Gupta, **SPLASH: Standford Parallel Applications for Shared Memory.** *Computer Architecure News,* 20(1):5-44, March 1922.

[46] K. E. Schauser, D. E. Culler, T. v. Eiken. **Compiler-Controlled Multithreading for Lenient Parallel Languages.** F*PCA '91 Conference on Functional Programming Languages and Computer Architecture,* Aug. 1991, Springer Verlag.

[47] D. Stein, D. Shah, **Implementing Lightweight Threads.** *Summer '92 USENIX*, San Antonio, Tx

[48] M. Steckermeier, F. Belosa, **Using Locality Information in User-level Scheduling.** TR-95-14. Computer Science Department, IMMD IV. University of Erlangen-Nurnberg, Germany.

[49] D. P. Stoutamire, S. Omohundro. **Sather 1.1 Specification.** International Computer Science Institute, Berkeley Ca. Technical Report TR-96-012.

[50] D. P. Stoutamire, M. Kennel, **Sather Revised: A High-Performance Free Alternative to C++.** *Computers in Physics,* Vol. 9, No. 5, Sep/Oct 1995 pp. 519-524.

[51] D. P. Stoutamire, **Zones: Portable, Modular Expressions of Locality**. Ph.D. Thesis. University of California at Berkeley, 1997.

[52] **The SPARC Architecture Manual. Version 8.** SPARC International, Inc., Prentice Hall, 1992

[53] **The SPARC Architecture Manual. Version 9.** Eds. D. L. Weaver, T. Germond. SPARC International Inc., Prentic Hall, 1994

[54] Sun Microelectronics. **UltraSPARC User's Manual,** 1996.

[55] Sun Microsystems. Products and Solutions: Solaris Products. An on-line document: http://www.sun.com/solaris/new/index.html.

[56] K Taura, S. Matsuoka, A. Yonezawa. **StackThreads: an abstract machine for scheduling fine-grain threads on stock CPUs.** *Theory and Practice of Parallel Programming. International Workshop '97*. Proceedings, pp. 121-136, Springer-Verlag, Berlin/New York, 1995.

[57] K. Taura, A. Yonezawa, **Fine-grain Multithreading with Minimal Compiler Support - A Cost Effective Approach to Implementing Efficient Multithreading Languages.** *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pp. 320-333. Las Vegas, June 1997.

[58] R. Thekkath, S. J. Eggers, **Impact of Sharing-Based Thread Placement on Multithreaded Architectures.** *In Proc. of the 21st Annual International Symposium on Computer Architecture.* Chicago, IL, April 1994. IEEE Computer Society Press, 1994.

[59] A. Tucker, **Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors.** Ph.D. Dissertation. Stanford University, December 1993.

[60] A. Tucker, A. Gupta, **Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors**, *The 12th Symposium on Operating Systems Principles,* pp. 159-166, December 1989.

[61] **The UltraSPARC Processor - Technology,** White Paper. Available from www.sun.com

[62] **The Ultra Enterprise 1 and 2 Server Architecture.** Technical White Paper. Sun Microsystems, April 1996.

[63] M. Vandevoorde, E. Roberts, **WorkCrews; An Abstraction for Controlling Parallelism**. *Int. J. Parallel Program.* 17, 4 (Aug. 1988), 347-366.

[64] D. B. Wagner, B. G. Calder. **Leapfrogging: a portable technique for implementing efficient futures.** *SIGPLAN Notices*, pp. 208-217, July 1993.

[65] B. Weissman, B. Gomes, J. W. Quittek, M. Holtkamp, **Efficient Fine-Grain Thread Migration with Active Threads**. Submitted to the *12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998)*

[66] C. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, K. Yelick. **Runtime Support for Portable Distributed Data Structures**, *Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, May 1995.

[67] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, **The SPLASH-2 Programs: Characterization and Methodological Considerations.** *22nd Annual International Symposium on Computer Architecture,* pp 24-36, June 1995