Active Threads: Enabling Fine-Grained Parallelism in Object-Oriented Languages

Boris Weissman, Benedict A. Gomes, Jerome A. Feldman
{borisv, gomes, jfeldman}@icsi.berkeley.edu
University of California at Berkeley and International Computer Science Institute
ICSI, 1947 Center St., Suite 600, Berkeley, CA 94704-1198
(510) 642-4274 x126, fax (510) 643-7684

Abstract

Many existing object-oriented languages and new language proposals include threads to achieve high performance on modern SMPs. This paper presents Active Threads, an extensible and portable high-performance thread system that addresses the runtime needs of OO languages. An event driven architecture capable of supporting different scheduling policies is presented. Thread scheduling that exploits the temporal and spatial locality is considered in detail. Active Threads is ported by retargeting the eight primitives of the Portability Interface. The system currently runs on several hardware platforms including SPARC, Intel i386 and higher, DEC Alpha AXP, HPPA and outperforms vendor thread packages as much as by orders of magnitude. Active Threads is used as a compilation target for Sather, a parallel object-oriented language developed at ICSI. Active Threads is also used by a threaded distributed extension of C++.

Keywords: threads, shared-memory multiprocessors, thread scheduling, locality, cache

1 Introduction

With advances in microprocessor technology, parallel platforms have become widely available. Moderately priced commodity SMPs are now manufactured by most major hardware vendors. Several programming models have been suggested to exploit the newly available processing power. Some of the object-based models focus on the formal semantics and the simplicity of the programming metaphor (Actors [1]), while others concentrate on execution safety (behavior abstraction [8]), or typing issues (Eiffel [12]). Many C and C++ dialects relying on vendor thread libraries have been developed to address irregular applications. Explicitly parallel object-oriented languages such as Java [2], and Sather [17] have also extended the sequential OO model with threads.

While several modern operating systems provide some support for threads and platform-independent thread standards are emerging (POSIX.1c [7], Java Threads [2]), most proprietary "native" thread systems remain too heavy-weight for fine-grain parallel programming. The majority of modern general-purpose thread packages remain adequate only for medium-grained concurrency and for improving the structure of applications with demands for interactive responsiveness (GUI).

1.1 The Goal: Fine-grained Concurrency

Many existing parallel languages support fine-grain parallelism. They 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 [3] and HPF [6] provide constructs to express fine-grain nested data parallelism. Control-parallel languages such as CC++ [5], Cilk [4], and Sather [17] support powerful control mechanisms for expressing fine-grained task parallelism.

Fine-grained object-oriented parallel programming on the emerging platforms has many important advantages:

- *Expressiveness*. A natural parallel decomposition of a program maps directly onto the threaded implementation.
- *Transparent load balancing.* With many threads, the task of load-balancing is off-loaded onto the threaded runtime.
- *Portability.* Fine-grained threaded programs are not

very sensitive to the number processors and can adapt to the changing number of cpus (both statically and dynamically).

- *Encapsulation of parallelism.* Efficient support for many threads (even millions) enables encapsulation of parallel patterns in reusable modules.
- *Hiding communication latencies.* Distributed platforms with SMPs as processing nodes can use fine-grained threads to mask communications latencies.

1.2 Granularity Constraints

In practice, the degree of parallelism that can be used effectively is constrained. The performance penalties come from several sources: 1) load imbalance; 2) thread management overhead (thread creation, synchronization, memory management); 3) locality effects (cache, TLB misses, paging, etc.)

Figure 1 demonstrates the relative importance of the granularity constraints. Load balancing, a factor of pre-

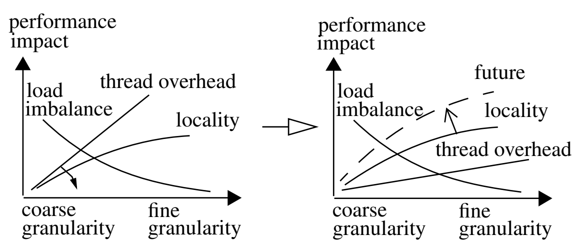

Figure 1: Granularity Constraints.

dominant importance for coarse-grained parallelism, has been extensively studied in the past [15]. With only a few threads, the penalty of load imbalance is high because it results in processor idle time (threads may block). As the granularity increases, the thread management overhead gains importance. The exact slope of the thread overhead curve is determined by the design and implementation of the runtime system. For instance, user-level threads are characterized by a smaller slope than kernel-level threads. For a given system, such overhead is usually fixed at a certain number of instructions (per thread or synchronization operation) and its significance grows linearly as thread granularity increases. Locality effects at all levels of the memory hierarchy (TLBs, on-chip cache, external cache, paging, etc.) also gain importance as thread granularity increases.

In an effort to reduce the thread management overhead, many researchers have turned to simpler parallel models. For instance, many runtime systems support only non-blocking threadlets that, once started, run to completion (Filaments [10], Cilk [4], Multipol [20]). While it is acceptable in some contexts (such as explicit continuation passing), the semantic restrictions that enable performance optimizations reduce the immediate usefulness of these techniques for explicitly parallel object-oriented languages which generally support very expressive threads and synchronization constructs.

While the issues involved in reducing the thread management overhead can be addressed by careful optimizations of thread and synchronization code, once these steps are taken, we are faced with a qualitatively different problem. An executing thread must get its working set into a processor's cache. The shorter the time the threads execute and the more threads block and get rescheduled, the greater the relative cost of such cache reloads. For 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 often relinquish the processor suffer from caching effects to a greater degree.

The trends of the last decade indicate that processors are getting faster at a much higher rate than memories: while microprocessor performance has been improving by 60% per year, the DRAM latency has been improving at less than 10%. Despite deeper and deeper cache memory hierarchies, the gap between the performance of processors and DRAMs has been increasing at 50% per year for 10 years [27][28].

In our diagram (right part of Figure 1), this widening of the performance gap corresponds to shifting the locality curve upward. The crossover point at which locality effects become predominant will be observed for coarser threads (and more applications) in the future than currently. Addressing the locality issues is fundamental to supporting effective fine-grained concurrency on current and future architectures.

2 Active Threads Architecture

This section describes the architecture of Active Threads, the general scheduling mechanism and specialized policies.

2.1 Overall Organization

Active Threads provides the following abstractions: a) *threads*, b) *thread bundles*, c) *virtual processors*, d) *synchronization objects*. Groups of logically related threads form thread bundles. Threads in the same bundle share a common thread scheduler. For instance, a common parallel loop statement can be implemented as a bundle of threads.

The overall Active Threads organization is shown in figure Figure 2. Unlike other systems, Active Threads distinguishes between the RUNNABLE and SCHED-

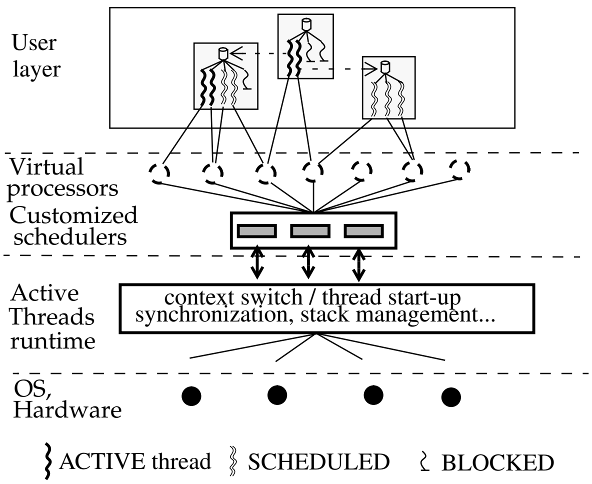

Figure 2: Overall Organization

ULED state to enable extensible schedulers. Any thread in a RUNNABLE state may be scheduled by a (possibly user-specified) scheduler.

Active Threads hide hardware dependencies such as the number of CPUs. Instead, the user is provided with a virtual processor abstraction. At the application level, threads can be scheduled to run on virtual processors. Active Threads clients are encouraged to schedule threads that are likely to share data to run on the same virtual processor. Such data dependent scheduling annotations are likely to produce substantial performance benefits (Section 4). This functionality can also be used to implement high-level OO abstractions such as zones [16] intended to capture locality patterns in a portable manner. No precise knowledge of the memory hierarchy is necessary, although some information may be beneficial.

Virtual processors are multiplexed over the available cpus. The application can change the mapping by supplying customized schedulers. A library of schedulers is provided and can be extended by the application.

A typical thread creation primitive is shown below

```
at_thread_t* at_create_x (at_bundle_t *b, int vproc,
                  at_userf_1_t *f, at_word_t a0, at_word_t
a1,...)
```

The thread creation primitive specifies the destination bundle, a virtual processor binding (including *unbound*), and usual information such as a function pointer and arguments.

The Active Threads runtime provides basic thread services: thread initialization, start-up and context switch, thread stack management. A variety of extensible synchronization objects (not shown) including spinning and two-phase locks, semaphores, barriers, and condition variables are supported. Much of the runtime is machine independent. Hardware dependent services are captured in the Machine-Dependent Layer.

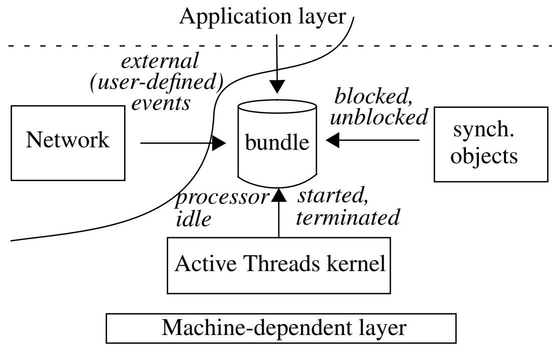

Figure 3: Scheduling Events

2.2 Scheduling Event Mechanism

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

A runtime instance of a bundle data structure consolidates all scheduling decisions for a group of threads. The bundle is free to maintain any scheduling data structures that fit most closely the semantics of the thread group. Different subsystem communicate with bundles by dispatching scheduling events. Bundles encapsulate all aspects of scheduling by providing event handlers for all scheduling events.

The relationship between bundles and other subsystems as well as the direction of the event flow are shown in Figure 3.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 are involved with less common and more specialized thread operations such as thread migration in distributed systems or committing threads to a persistent store. 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 are given in [18]. All internal events and their logical origins are listed in Table 1.

Normally, threads in the ACTIVE (executing) state continue running until they block on synchronization objects, yield execution or terminate. If this happens, the runtime performs several steps before dispatching an event to a thread's bundle. 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 dif-

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 (sync.object)	informs about thread unblocking.
bundle created (kernel)	informs about a new child bundle.
bundle terminated (kernel)	informs about termination of a child.
processor idle (kernel)	requests more threads for dispatching by the idle processor.

Table 1: Internal Events.

ferent 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. After the elementary thread operations are completed, a scheduling event that contains a thread handle is dispatched to an appropriate bundle. This allows the bundle to handle threads as abstract data types and perform scheduling decision in a machine-independent manner. For instance, a simple handler for a *thread unblocked* event may immediately schedule the unblocking thread for execution:

```
void thread_unblocked_h(at_bundle_t *b, at_thread_t
*t) {
    /* Just schedule thread on cpu returned by map(t)*/
    at_schedule(map(t), t);
}
```

A more complex scheduler may keep an unblocking thread if higher priority threads are available. More details and practical examples are given in [19].

2.3 Scheduling Policies

Bundles implement specialized scheduling policies by supplying handlers for scheduling events. The runtime uses the event mechanism to inform the bundle about state changes of its threads. The bundle is free to react to these changes in any way. Active Threads is distributed with a library of schedulers including several versions of LIFO, FIFO, and locality schedulers. Clients may extend the scheduling library [19].

3 Portability

The machine-dependent layer hides the hardware details of modern SMPs. Porting Active Threads to a different architecture involves retargeting the machine-dependent layer by implementing the Portability Interface on the intended platform.

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.The Portability interface primitives are described in Table 2

Primitive	Semantics	
initialize	initialize the machine-dependent subsystem	dozens of C lines
has-private-storage?	does platform supports a 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	few assembly lines
number of cpus	return the number of currently available physical processors	a dozen C lines
thread context switch	switch between two thread contexts: stopping the thread, and saving and reloading registers.	a few dozen assembly lines
thread initialize	initialize thread's context with a function address and arguments.	few assembly lines

Table 2: Active Threads Portability Interface

4 Performance

4.1 Microbenchmarks

We have measured the performance of Active Threads on different models of SPARC symmetric multiprocessors, Intel Pentium Pro, DEC Alpha AXP, and HPPA 1.1. Performance of Active Threads relative to vendor threads on these platforms is shown in Figure 4. This comparison is important since "native" thread systems are often chosen as compilation targets for concurrent object-oriented languages

The benchmark operations are the following.

- *thread create*: the creation of a new thread, including stack allocation, but excluding a context switch and execution;
- *null thread*: execution of a thread performing a null call;
- *context switch*: a thread repeatedly yields to another thread
- *uncontested mutex and uncontested semaphore*: times successful mutex 'lock' and semaphore 'wait' operations;
- *mutex and semaphore try*: the overhead of the non-blocking versions of the corresponding operations;
- *mutex and semaphore ping-pong:* repeatedly synchronize two threads with one another in a way used to measure the synchronization cost of Solaris Threads in [13].

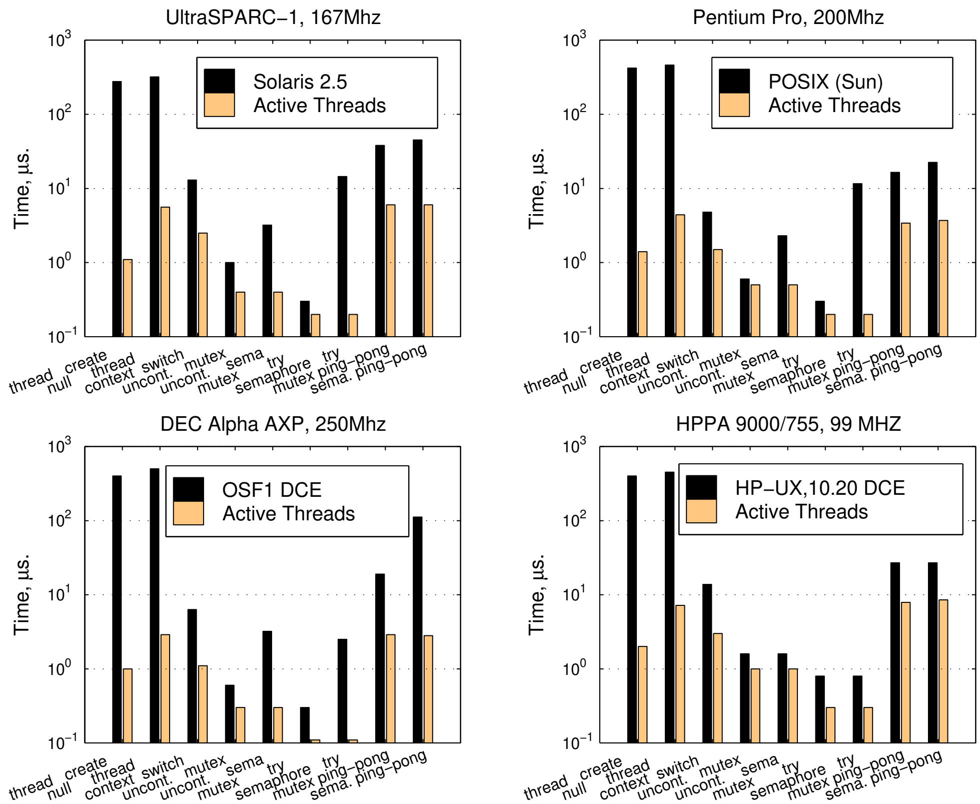

Figure 4: Active Threads vs. proprietary threads.

By comparison, a null procedure call on the UltraSPARC-1, model 170 takes 0.75µs when register window overflow occurs and 0.08µs without window overflow. Thread creation overhead on this platform is 1.3µs.

4.2 Applications and Locality Effects

To quantify the benefits of customized schedulers, we have implemented two fairly simple parallel applications: quicksort that switches to bubble sort for leaf nodes and mergesort that switches to insertion sort for leaves. Both implementations employ a fine-grain decomposition - separate threads are created for each recursive subdivision of the input. Our implementation of quicksort uses an array based representation while mergesort deals with the linked list representation. [9] evaluates several runtimes including Active Threads for conservative parallel manufacturing simulations.

Scheduling Policies and Performance

The same application sources were compiled and linked with several different thread schedulers: 1) FIFO; 2) FIFO with MCS (memory-conscious scheduling); 3) FIFO with lazy thread stack allocation; 4) FIFO with lazy stack allocation and MCS; 5) LIFO; 6) LIFO with MCS; 7) LIFO with lazy thread stack allocation; 8) LIFO with lazy stack allocation and MCS.

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 (after Markatos and LeBlank [11]) 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. With lazy versions of these policies, threads are bound to stacks when they are dispatched for the first time instead of at creation time.

Figure 5 presents performance results for mergesort obtained on the Sun Enterprise 5000 with eight 167Mhz UltraSPARC-1 processors (input of 100,000 records with the threshold 10 to switch to elementary sorts.) Figure 6 provides similar information for quicksort.

By departing from the standard FIFO policy, we can speed up mergesort by a factor of 6 on the 8cpu E5000.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 branch-and-bound applications.

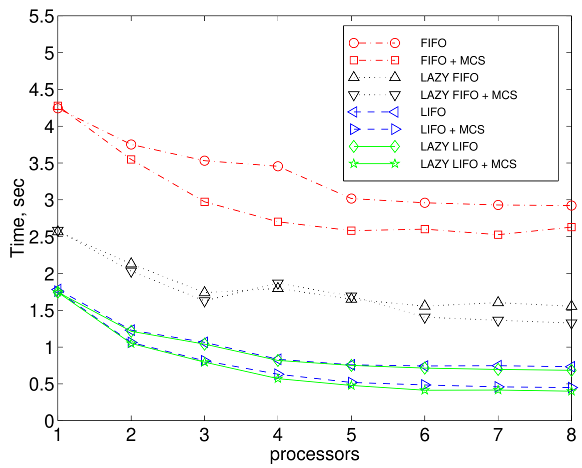

Figure 5: Mergesort Performance.

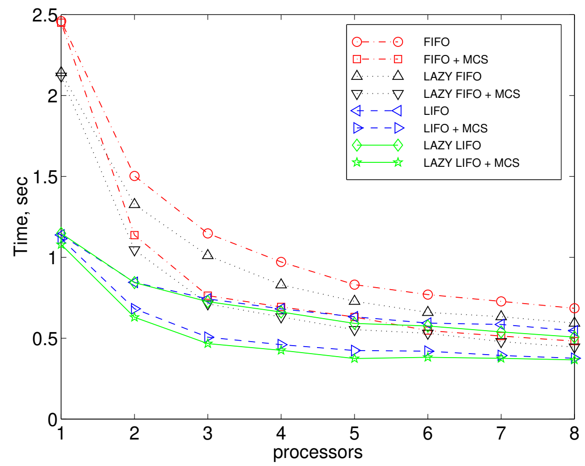

Figure 6: Quicksort performance.

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* supported scheduling policy.

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 in the FIFO case (FIFO roughly corresponds to a breadth-first execution of a tree). Fewer runnable threads require less memory for stacks.

Lazy stack allocation improves performance for similar reasons. It also adds an additional locality benefit. If stacks are allocated at thread start-up, stacks from recently terminated threads that are likely to be cached are recycled immediately.

Performance Impacts of the Memory Hierarchy

While moving from FIFO to LAZY FIFO and from FIFO to FIFO+MCS better performance is achieved by exploiting different levels of the memory hierarchy. The LAZY FIFO version outperforms our original FIFO version mainly because of the virtual memory effects. LAZY FIFO is characterized by reduced memory usage due to better thread stack reuse. Such reduction results in fewer page faults and lower OS trap handling overhead. Figure 7 repeats the runtime curves for quicksort using FIFO, LAZY FIFO and FIFO+MCS schedulers. The right part of the figure shows the number of page faults for the three considered scheduling policies. Better performance of LAZY FIFO relative to FIFO is mostly due to a smaller number of page faults. However, better performance of FIFO+MCS relative to the original FIFO is due the cache effects. As Figure 7 shows, FIFO and FIFO+MCS versions incur roughly the same number of page faults. However, FIFO+MCS reuses the individual processor's cache significantly better. In the quicksort context, cache effects are more important than the VM effects and FIFO+MCS outperforms LAZY FIFO in spite of the higher memory usage.

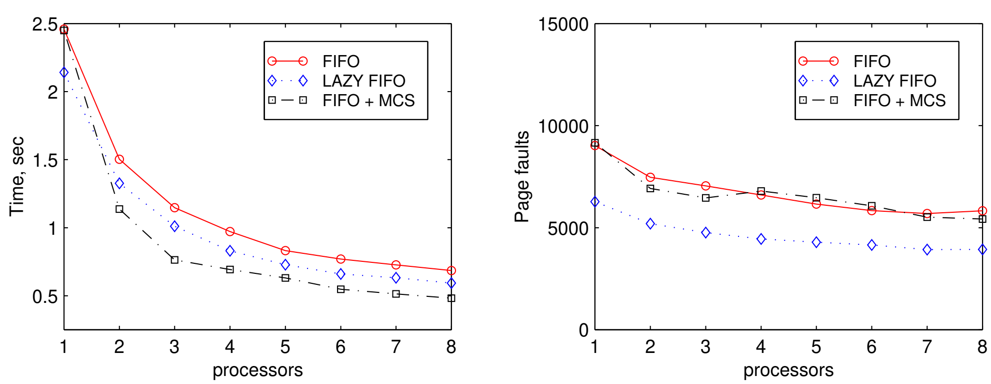

Figure 7: Quicksort: page faults vs. cache effects.

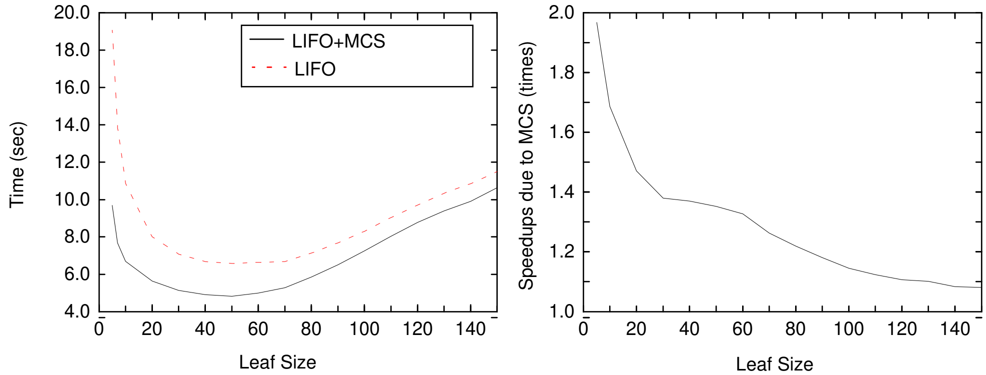

Figure 8: Memory-access patterns.

Locality Scheduling and Memory Access Patterns

Our sorting examples provide a good testbed to investigate the effects of MCS on applications with different memory access patterns. 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 it. For large leaf sizes n, a penalty of the initial cache miss is amortized over a large number of operations. By varying the size of leafs, we can simulate different memory access patterns.

The left part of Figure 8 shows how the overall performance depends on the leaf size (the base algorithm is LIFO). The right part displays how the benefits of MCS depend on the regularity of memory accesses. 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 MCS are the largest for very small leaves (irregular memory patterns). The relative speedup due to MCS falls for large leafs because of the amortization of cache misses. The larger the leaf size n, the less the relative effect of the initial misses. 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 cost of cache misses.

Therefore, *speedups* should be greater for larger leaves, which is indeed the case (Figure 9.) However, it is a well established fact that serial quicksort *performs* the best for mid-range leaf sizes [14]. This is also true for our simple parallel version. MCS alone brings significant improvements for mid range leafs (up to 40%).

The quicksort example illustrates the well known fact that speedups are not always the best measure of overall performance. The speedup curve corresponding to the highest overall performance corresponds is a solid line curve in Figure 9.

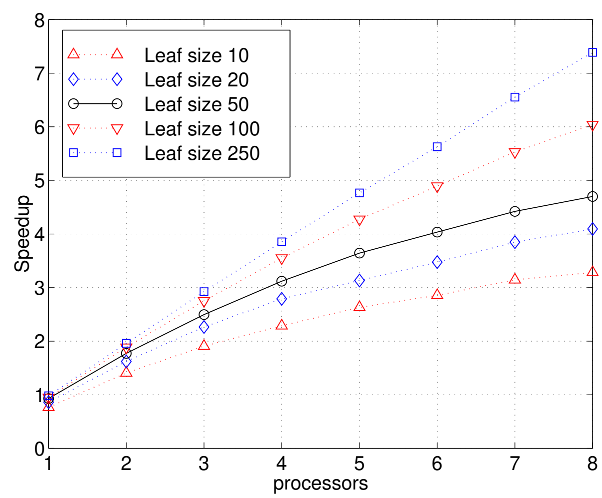

Figure 9: Quicksort speedups.

5 Conclusion

Active Threads is a portable high-performance runtime for modern OO languages. Efficiency is achieved by optimizing thread and synchronization primitives and addressing the locality issues. Applications may extend the runtime by providing their own thread schedulers. The system has been ported to several platforms and substantially outperformed native thread systems. Active Threads is used as a compilation target for Sather[1] and a distributed threaded extension of C++.

References

[1] G. Agha, Actors, A Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

[2] D. Berg. Java Threads, A Whitepaper. Sun Microsystems, March 1996.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] D. G. Kafura and Keung Hae Lee. Inheritance in Actor based concurrent object-oriented languages. ECOOP'89, pp. 131-145. Cambridge University Press, 1989.

[9] C. C. Lim, Y. H. Low, W. Cai, W. Hsu, S. Y. Huang. An Empirical Comparison of Runtime Systems for Conservative Parallel Simulation. 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP), in conjunction the IPPS/SPDP 1998, March 1998.

[10] 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

[11] 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 Scientific Publishing, 1994.

[12] B. Meyer, Systematic Concurrent Object-Oriented Programming, Communications of the ACM, September 1993, Vol. 36, No. 9, pp. 56-80.

[13] 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.

[14] R. Sedgewick, Algorithms. Addison-Wesley, 1988.

[15] N. G. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. Computer, vol. 25, no. 12, Dec. 1992, pp. 33-44.

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

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

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

[19] B. Weissman, Active Threads: an Extensible and Portable Light-Weight Thread System, ICSI TR-97-036, Oct. 1997.

[20] 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.

1. Active Threads is available as a part of the Sather distribution from www.icsi.berkeley.edu/~sather