Performance Counters and State Sharing Annotations: a Unified Approach to Thread Locality

Boris Weissman
University of California at Berkeley and International Computer Science Institute
1947 Center St, Suite 600, Berkeley, CA 94704
borisv@icsi.berkeley.edu

Abstract
This paper describes a combined approach for improving thread locality that uses the hardware performance monitors of modern processors and program-centric code annotations to guide thread scheduling on SMPs. The approach relies on a shared state cache model to compute expected thread footprints in the cache on-line. The accuracy of the model has been analyzed by simulations involving a set of parallel applications. We demonstrate how the cache model can be used to implement several practical locality-based thread scheduling policies with little overhead. Active Threads, a portable, high-performance thread system, has been built and used to investigate the performance impact of locality scheduling for several applications.

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. To enable access to the newly available processing power, many existing programming languages were extended with threads and many new language designs based on threads were proposed. Such languages usually 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. Many languages such as Java [4], Cilk [6], or Sather [24] support powerful mechanisms for expressing task parallelism.

Fine-grained thread programming has many important advantages such as a natural parallel decomposition of a problem, transparent load balancing and portability across platforms with different numbers of processors.

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, etc.); 3) locality effects (cache, TLB misses, paging, etc.). Load balancing, a factor of predominant importance for coarse-grained parallelism, has been extensively studied in the past. Past studies also demonstrated that the context switch overhead of general purpose blocking threads can be within an order of magnitude of a function call cost [1].

While the issues involved in reducing the thread management overhead can be addressed by careful optimizations of thread and synchronization code, locality effects present a qualitatively different problem. An executing thread must get its working set into a processor's cache. The more fine-grained the threads are and the more they 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 block frequently 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%. The gap between the performance of processors and DRAMs has been increasing at 50% per year for 10 years [18]. The cost of the second-level cache misses for modern machines is high. For instance, on our implementation platform, a Sun Enterprise 5000 server with 167Mhz UltraSPARC-1 processors, the E-cache miss takes up to 80 cycles. A 300Mhz DEC Alpha and 200Mhz SGI MIPS R10K systems have main memory latencies of 0.4µs and 1.1µs respectively (or 120 and 220 cycles)[16] [8]. On such platforms, the cycle cost of a thread context switch can match that of a single secondary cache miss [32].

Given the increasing processor-memory performance gap, addressing the locality issues at all system levels (architecture, OS, compilers and runtime systems, language design, user annotations), becomes fundamental to supporting effective fine-grained concurrency. In particular, thread scheduling that maximizes cache reuse is the key to achieving efficient fine-grained parallelism.

This paper describes a combined approach for improving locality that uses the performance monitors of modern processors and user annotations to guide thread scheduling on the multiprocessor. The described system is a part of a larger parallel object-oriented language design and implementation effort and the presented techniques have been used in the Sather compiler and runtime system [24]. Such a context imposes several requirements for a practical solution: it must handle general-purpose blocking threads; it must be sufficiently lightweight to address the dynamic/irregular problem domain; it must not violate OO encapsulation. The last constraint is violated, for example, if the programming model allows the use of system-centric annotations such as processor numbers to influence the placement of threads and objects.

This paper makes the following contributions:

- We present an analytical cache model for blocking threads in the presence of data sharing. The model predicts the size of the cached thread state on-line, as the computation unfolds by using a) the on-line cache access information accumulated by many modern processors; b) user annotations that reflect thread data-sharing patterns inherent in the application.
- We evaluate the accuracy of the proposed model by detailed simulations involving a set of parallel applications.
- We show how the cache model can be applied to implement several practical locality scheduling policies with little overhead.
- We have designed and implemented Active Threads, a portable

parallel runtime system, and used it to evaluate the performance impact of several locality scheduling policies on modern platforms. Active Threads has been also used as compilation target for Sather, a high-level object-oriented language. Our results indicate the elimination of a substantial number of the secondary cache misses and significant performance improvements.

2 The Shared State Model

In this section, we describe the analytical cache model for threads in the presence of data sharing. We start by defining terminology and the underlying assumptions. We next describe the input parameters to the model: information obtained from the hardware performance monitors and user program annotations. Finally, we derive the model for large direct-mapped secondary caches.

2.1 Terminology and assumptions

An executing thread must get its working set into a processor's cache. In particular, a thread scheduled for the first time or an unblocking thread usually experiences a burst of cache misses to establish or restore its working set in the cache. We call such misses *the cache-reload transient* after Thiébaut and Stone who first introduced this term in the context of multiprogramming [31]. We also use their terminology to designate the lines in the cache that belong to a particular thread as this thread's *footprint* in the cache. A footprint in the cache can be thought of as a projection of a thread's working set onto the cache. In the context of this paper, we will use the term footprint to mean such a projection.

Thiébaut and Stone assumed *F*, the size of a program footprint in lines, to be known and it was used as an input parameter to drive their task interaction model. Agarwal et al. noted that no method to obtain such footprints was given and indicated that it could be inferred by analyzing collected program traces off-line [2].

Falsafi and Wood describe a scheme for extracting the sizes of process footprints in the cache from several parallel runs while flushing the cache before some executions [9].

Our goal is to design an analytical model that will predict the footprint size on-line for each thread/processor combination as the computation unfolds. This information can, in turn, be used to guide thread scheduling by a selected scheduling policy. Similar to the recent work of Bellosa et. al [3], the expected footprint size *F* is the output of our analytical model and is used to guide thread scheduling. We aim at a simple model that can be evaluated on-line at the thread context switch time.

We make a relatively common assumption that accesses to cache are independent and uniformly distributed [2][31][21][9]. Factors contributing to the validity of this assumption include the hashing operations mapping large address spaces to a smaller number of cache lines [2] and heuristics used by the VM systems to map virtual pages to physical frames. Another contributing factor is the relatively large lines of secondary caches on modern platforms. As noted in [2], *run* lengths (streams of sequential references) generally range from one to ten words and therefore can often be accommodated entirely by large secondary cache lines. Our assumption is less valid for programs that exhibit very long run lengths and for architectures with such relatively uncommon features as virtual secondary caches.

The model specifically addresses large off-chip physical direct-mapped caches that are able to accommodate *many footprints*. Such secondary caches are common on modern SMPs: E-cache of Sun Enterprise servers (up to 4Mb) [30], B-cache of DEC AlphaServer 4100 (up to 4Mb) [23], external cache of HP Exemplar (1Mb) [10]. The developed model can be extended to the associative cache case (although the analytical results are likely to be more complex with a higher runtime overhead).

2.2 Hardware performance monitors

Many modern processors used as building blocks of SMPs already collect cache use information in on-chip registers. For instance, the UltraSPARC processor is equipped with two 32-bit Performance Instrumentation Counters (PICs) that can be configured to measure secondary cache references and hits in user mode, supervisor mode, or both [27]. Moreover, user-level access to these registers is enabled by setting a bit in the Performance Control Register (PCR) providing the runtime systems with "free" cache use information.

In a similar vein, the Pentium Pro processor provides two 40-bit performance counters PerfCtr0 and PerfCtr1 that can be read at any privilege level and configured to count the number of secondary cache misses [12]. The IBM POWER2 [34], HPPA 8000, and MIPS R10000 systems also include performance monitoring units.

While the actual events measured by performance counters vary on different architectures, it is usually possible to either count the number of secondary cache misses directly, or reconstruct this number given the information monitored by the counters. Our cache model makes a minimal assumption of the availability of the number of cache misses that happen between two thread scheduling points. The model uses this number as an input parameter to predict thread footprint sizes for affected threads at the context switch time. User annotations, the second kind of input to the model, are described in the following section.

2.3 Programming model and user annotations

We address the general unrestricted thread programming model: threads are units of (possibly parallel) execution with independent lifetimes and separate stacks that share the address space and other resources. Threads may block by performing synchronization operations involving any of the common synchronization objects: mutual exclusion locks, semaphores, barriers, condition variables, etc. POSIX threads [11], Java threads [4], or a number of other threaded languages fit this general description.

In the thread programming model, the thread abstraction may be assumed to carry some information about locality of thread's state. However, threads alone cannot express reference locality patterns that exist between different threads.

We extend the thread programming model with program-centric user annotations that specify thread state-sharing patterns inherent in the applications. Such annotations are just hints to the runtime system - the more detailed they are, the better, but incomplete or incorrect annotations do not affect the correctness of code.

Consider threads a and b in Figure 1. Shaded areas enclosing the threads represent the state actively accessed by the threads at some point in time.

A traditional thread abstraction fails to capture the fact that the thread state may be partially or fully shared with other threads. We extend the traditional model by adding annotations to express the sharing patterns. Using the syntax of Active Threads, such

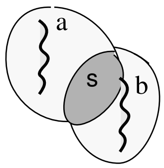

Figure 1: Thread state sharing.

annotations for the above example may be

```
/* tid_a, and tid_b are thread ids of the two threads */
/* a and b are thread state sizes, s is shared state size*/
at_share(tid_a, tid_b, s/a);
at_share(tid_b, tid_a, s/b);
```

Calls to *at_share()* specify what portion of one thread's state is shared with another thread. Consider another, more practical example. The (somewhat simplified) code fragment below performs simple parallel mergesort: the input list is split into left and right sublists to be sorted by separate threads. The sorted sublists are then merged by the parent thread.

```
tid_l = at_create(merge_thread, (at_word_t) left);
tid_r = at_create(merge_thread, (at_word_t) right);
join(tid_l); join(tid_r); /* wait for chid threads to finish */
merge_sublists(left, right);
```

Figure 2 illustrates the sharing pattern induced by this code.

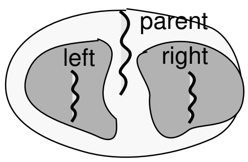

Figure 2: Mergesort: thread state sharing.

To express such sharing, the program may be annotated by inserting these lines in the parent thread code following thread creations (*at_self()* returns the calling thread's id)

```
at_share(tid_l, at_self(), 1.0);
at_share(tid_r, at_self(), 1.0);
```

The annotations reflect the fact that the state of child threads is fully contained in the parent thread's state. While the shown annotations are incomplete in the sense that they do not describe the relationship between generations of threads more than a level apart (we do not assume any transitive properties of annotations), they may be adequate to achieve substantial performance gains as they capture first-order effects. Also note that in the example the parent thread prefetches no data for the children and the corresponding calls to *at_share()* are omitted.

More formally, user annotations specify a directed shared state dependency graph $G=(V,E)$ and sharing coefficients $q_{i,j} \in [0,1]$ associated with each arc $(t_i,t_j) \in E$, where $t_i,t_j \in V$ are runtime thread instances. At each point in time, the value of $q_{i,j}$ specifies what portion of the state of thread t_i is shared with the state of thread t_j. We say that the destination nodes of edges are *dependent* on the source nodes (the cached state of t_j depends on activity of t_i).

The graph is created dynamically as the computation unfolds. Executing the code associated with a user annotation results in an edge being added to the graph at runtime or a weight change if the edge already exists. We can also think of graph G as a complete

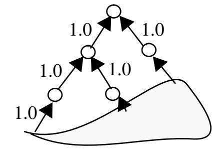

Figure 3: A state dependency graph induced by partial annotations of mergesort.

graph with unspecified edges having 0 coefficients.

For instance, user annotations for our mergesort example may give rise to the snapshot of G shown in Figure 3. We do not assume any transitive properties of the annotations. Also, as the mergesort example indicates, the edges do not need to be always bidirectional.

2.4 The shared state cache model

We now develop an analytical cache model that, for each thread, predicts the size of its state in the cache of each processor as the parallel computation unfolds. The shared state model takes as input the number of cache misses during a scheduling period reported by the performance monitoring hardware and a current shared state dependency graph induced by the user annotations to predict the size of thread footprints in each processor's cache.

As mentioned earlier, the model is tailored for the large physical direct-mapped secondary caches. Let N be the size of such cache in lines, and let F denote the size of the footprint in the cache in lines that the model must predict.

Consider thread A blocking on processor p at time t. While running on processor p during the last scheduling period, thread A has taken n misses as reported by performance counters. Our goal is to compute the effect of these misses on the footprints of all threads (including A) in the cache of processor p.

We divide the set of all threads into 3 disjoint subsets: 1) blocking thread A itself; 2) threads independent of A (threads that do not share state with A; all nodes in G that are not destination of edges starting at A); 3) threads dependent on A (threads that share state with A; the destination nodes of edges in G starting at A).

Such a partition will also come useful in Section 4 when we present several practical scheduling frameworks.

We proceed by considering three threads, A, B, and C from the three thread classes respectively. We assume that when thread A was last scheduled for execution before blocking, its original footprint was S_A lines and the footprints of threads B and C were S_B and S_C lines respectively. These numbers are known at time t.

Case 1: Blocking Thread A

Consider some cache line L that is *not* in A's footprint. Suppose thread A takes a cache miss. The new line may displace any of the N lines in the cache. The probability that it will land in one of the lines other than L is $P_1 = \frac{(N-1)}{N}$. After n misses, the probability that the considered line is still not in A's footprint is

$$P_n = \left(\frac{N-1}{N}\right)^n$$

The original number of cache lines that are not in A's footprint is $N - S_A$. The expected number of such lines after n misses is

$$E_n[\tilde{F_A}] = (N - S_A)P_n = (N - S_A)\left(\frac{N-1}{N}\right)^n$$

Finally, the expected size of A's footprint after n misses is:

$$E_n[F_A] = N - E_n[\tilde{F_A}] = N - (N - S_A)\left(\frac{N-1}{N}\right)^n$$

A similar expected value is obtained by a different argument by Bellosa at el. in [3].

Case 2: Independent thread B

Thread B can be in any of the following states: ready for execution, but not scheduled, blocked on a synchronization object, or executing on a different processor. Because of the activities of thread A, B's footprint in the cache of processor p will decay.

As before, we consider a single miss taken by thread A. However, now we concentrate on the line L that contains data for thread B rather than A. The probability that a single miss by thread A does not displace this line is $P_1 = \frac{(N-1)}{N}$. After n misses, this probability is

$$P_n = \left(\frac{N-1}{N}\right)^n$$

Since B originally had S_B lines in cache, the expected size of B's footprint after n misses by A is given by:

$$E_n[F_B] = S_B\left(\frac{N-1}{N}\right)^n$$

Case 3: Dependent thread C

This analysis is more complex and is presented fully in the Appendix. The analysis involves many different cases and to simplify the derivations, we model the cache behavior of thread C with a Markov chain. A closed-form solution for the expected size of the footprint of C is given by

$$E_n[F_C] = q_{A,C}N - (q_{A,C}N - S_C)\left(\frac{N-1}{N}\right)^n$$

where $q_{A,C}$ is a weight of (A,C) in the dependency graph G. If we substitute $q_{A,C} = 1$, i.e. complete inclusion, into the closed-form solution for C, we will obtain a solution for case 1). Alternatively, substituting $(q_{A,C} = 0)$ (threads have no shared data), will yield a solution for case 2).

Our model does not take into account invalidation effects when data cached by one processor is modified by another (for instance, if a thread migrates or a dependent thread is dispatched on a different cpu). We will consider the shortcomings in Section 3.4.

3 Model Evaluation: Simulations

To evaluate the accuracy of the model, we have performed simulations involving several applications written in C with direct calls into the thread library, and also applications written in Sather, a locally designed and implemented parallel object-oriented language. The purpose of the simulations is to observe the thread footprints in the cache for unfolding computations and compare them with predicted values. Such observations are impossible to make by monitoring running applications using only the hardware performance counters of modern processors. While the raw hit/miss statistics can be accumulated, the information about the association between cache lines and threads is lost. Hardware simulations that preserve such association are necessary. We have implemented a cache simulator that maintains the needed information in order to observe the changing thread footprints and compare them with the predicted values.

3.1 The simulated platform

We simulate the UltraSPARC model 170 with the parameters shown in Table 1 [28][29]. The UltraSPARC platform is selected for several reasons. First, its memory hierarchy does not have unusual properties such as virtual second-level caches. Secondly, the UltraSPARC processor is used in several lines of commercially available SMPs. Finally, the UltraSPARC processor has the necessary performance diagnostic hardware to explore different locality scheduling policies. Our performance results of Section 5 are also obtained on the serial and parallel platforms using the processors with characteristics in Table 1.

I-cache	16 Kb, 2-way associative, physically indexed and tagged, 32 byte line.
D-cache	16Kb, direct-mapped, virtually indexed and physically tagged, 32 byte line with two 16 byte sub-blocks, write-through, non-allocating.
E-cache	Unified external (L2) cache, 512Kb, physically indexed and tagged, 64 byte line, write-back, maintains inclusion for both I-cache and D-cache, hit time 3 cycles, miss penalty 42 cycles (for Enterprise 5000, miss penalty is 80 cycles if the line is cached by another processor, and 50 cycles otherwise)

Table 1: Simulated UltraSPARC-1 memory hierarchy.

Our cache/thread simulator is based on Shade, a fast instruction set simulator [7]. A SPARC V9 version of Shade is currently supported by Sun Microsystems [26]. Shade simulates the execution of instructions of the unmodified application binaries by dynamically compiling the executable code for the target machine into the executable code that runs directly on the host. Shade merely simulates the instruction set. However, it provides the necessary hooks to forward the needed instructions and data to other custom built simulation units such as the cache simulator.

In our case, all memory references and other information necessary to enable per-thread tracing are directly forwarded from Shade to our own cache simulator without accesses to non-volatile storage or IPC. Shade and the cache simulator run in the same process. We have designed the cache simulator to "understand" Active Threads context switches and accumulate per-thread statistics for different levels of the memory hierarchy (I-cache, D-cache, unified E-cache).

We are mostly concerned with the external (L2) E-cache of the UltraSPARC. The UltraSPARC E-cache is physically indexed and physically tagged, while the addresses generated by Shade are virtual. We have implemented a variant of the hierarchical page mapping policy suggested by Kessler and Hill [13] to simulate address translation. This policy tries to select a virtual to physical page mapping at page fault time that is likely to reduce cache contention and was shown to perform better than a naive (arbitrary) page placement.

3.2 Microbenchmarks

To illustrate the basic model, we have performed a series of microbenchmarks (applications are considered in the following section). The "main" thread in these simulations performs a

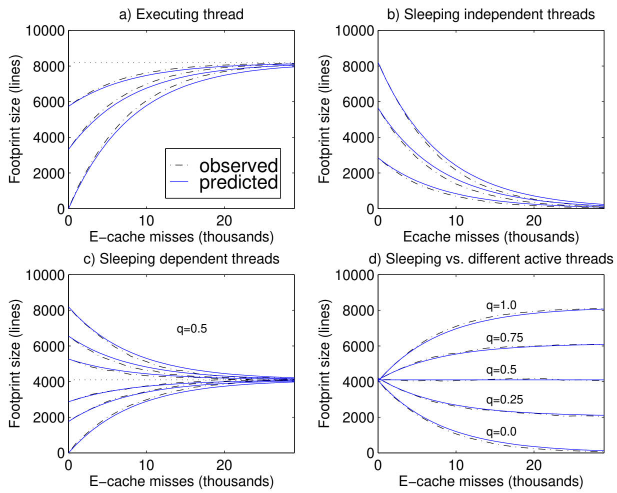

Figure 4: Random memory walk.

random memory walk. The simulations trace the footprint of the main thread and also the impact of its cache misses on the footprints of other threads. In particular, we study the decay of the footprints of sleeping threads that are independent of the main thread and the behavior of the footprints of dependent threads. The simulations are performed for a single processor UltraSPARC-1.

Part a) of Figure 4 shows the predicted and observed sizes of the executing main thread footprint as function of the number of E-cache misses (different curves correspond to different initial footprint sizes). Part b) illustrates the decay of the cached state of independent sleeping threads as the random walk thread executes. Parts c) and d) deal with dependent threads whose state partially overlaps that of the executing main thread.

Part c) illustrates the behavior of a sleeping thread, half of whose state is shared with the executing thread. Different curves correspond to different initial footprints. Depending on its initial size, the footprint may either decay or increase as shown in the figure. Finally, part d) of Figure 4 shows different scenarios for the change in a single sleeping dependent thread's footprint. Different curves correspond to different sharing coefficients with the executing threads. Once again, depending on the initial footprint and the state sharing coefficient, the footprint of the sleeping dependent thread may either increase or decrease due to misses taken by the executing (main) thread.

It is not very surprising that the microbenchmarks demonstrate excellent correspondence between the observed footprints and those predicted by the model since the model is based on the independence of references assumption exhibited by the random memory walk. In the following section we evaluate the accuracy of the model for actual workloads that may not be in good agreement with the underlying assumptions. We would also like to see to what extent the independence of references assumption is violated in applications written in C and object-oriented languages.

3.3 Applications

The applications used in the simulation are shown in Table 2. The top 4 applications are from the SPLASH-2 parallel application suite [35]. The application sources are used unmodified. The UltraSPARC binaries were obtained by implementing an Active Threads version of the PARMACS interface [14], a portability layer of the SPLASH-2 suite. We used the default data sets and

fmm	simulates a system of bodies over a number of timesteps using the adaptive Fast Multipole Method: unstructured patterns for communications and thread data distribution.
volrend	renders a three dimensional volume using ray casting
barnes	similar to fmm, but uses the Barnes-Hut hierarchical N-body method. The SPLASH-2 version improves data locality by the careful implementation of the oct-tree cell structures.
raytrace	renders a 3D scene using ray tracing. Each contact results in multiple rays.
particles	a naive N^2 simulation of a system of frictionless spheres of fixed radius possibly colliding particles.
photo	a simple photo-retouching program. Separate per color filters are applied to a pixel and its neighbors to compute new rgb values.
merge	parallel mergesort that switches to insertion sort for small sublists. The input data is uniformly distributed.
typecheck	the Sather compiler type checker thread: walks the abstract machine tree checking type conformance of nodes

Table 2: Simulated workloads

input parameters as recommended by the SPLASH-2 suite.

The bottom four applications are implemented in Sather, a locally designed high-performance parallel object-oriented language that provides a rich set of thread and synchronization constructs [19]. One of our goals is to study the influence of object-orientation and the associated compiler technology on the cache behavior. We have measured the footprint sizes of the "work" threads in each application after the initialization stage completed. The "work" threads are blocked during the computation stage and their state is flushed from the cache. After threads resume, their footprints are monitored by our cache simulator. As before, we monitor the uninterrupted execution of a single "work" thread on an UltraSPARC-1 processor.

Figure 5 shows the predicted and observed footprints as function of the number of E-cache misses for 6 out of 8 considered applications (the remaining data are presented in the following section). Figure 6 reports the E-cache performance in misses per instruction (MPI).

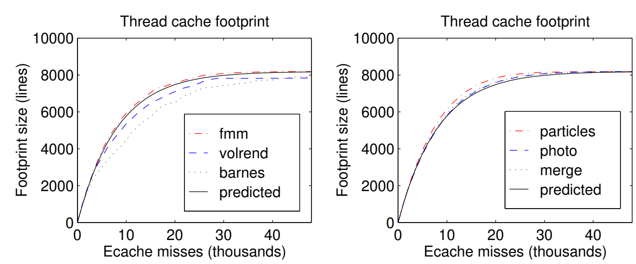

Figure 5: Observed footprints versus predictions.

As Figure 6 illustrates, unblocking threads usually experience bursts of reload transient misses followed by a period of a relatively stable number of misses.

For most applications, observed footprints are in good agreement with the cache model predictions. For applications written in C, the predicted footprints are somewhat larger than those observed by the simulator. We believe, the difference is due to higher clustering of references than that expected by the model (by the

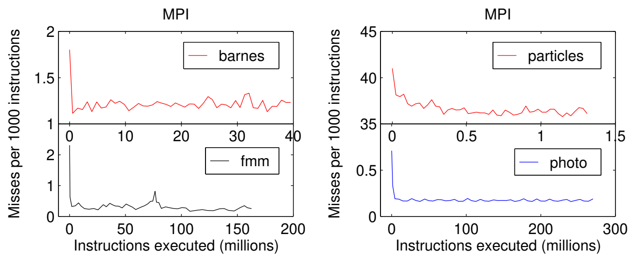

Figure 6: Average E-cache misses per 1000 instructions.

independence of references assumption). For instance, *barnes* was specifically optimized for locality in the second release of SPLASH and the predicted footprints for barnes are somewhat higher than observed.

Because of the extensive use of linked structures, OO programs tend to demonstrate less clustering of references than programs written in C. In our limited experiments, there was generally good correspondence between the predicted and observed footprints for Sather programs.

3.4 Shortcomings of the model

For two considered applications, the footprints in the cache predicted by the model were substantially larger than those observed by the simulations (Figure 7). We believe, the difference is explained by the specific semantics of the simulated threads.

The Sather typechecker thread is characterized by a fairly large working set - the type graph including the subtyping information for the entire compiled source tree (in the trial runs, the Sather compiler was compiling itself which leads to a fairly large type graph). The unblocking thread initially experiences a very intensive burst of misses as the type graph is brought into cache. The typechecker thread walks the abstract machine tree and performs semantic analysis for each node with the help of the type graph. The abstract tree is traversed in the order of creation which causes long run lengths and high clustering of cache references. The pattern displayed by the typechecker thread is similar to what Agarwal et al. call nonstationary behavior of programs performing the same sort of operations on each element of a large data set [2].

After the initial burst, the typechecker thread experiences a relatively small number of misses per instruction. This can be detected on-line if the runtime system monitors MPI as well as the number of misses for each thread. At this point, the runtime may attempt to switch to a different prediction heuristic. However, more experiments are needed to justify the increased complexity of the runtime system and to identify suitable heuristics.

Raytrace also demonstrates anomalous behavior. In between short bursts, the majority of misses are conflict misses that do not significantly increase the footprint.

There is another reason for the limited accuracy of the model. Our model does not address invalidation transactions when data cached on one processor is modified by another. While the model can be expanded to include the invalidation effects, in practice the performance instrumentation counters of the hardware available to us could not keep track of the secondary cache misses and invalidation events at the same time. We therefore avoid the extra complexity since it brings no benefits in our practical setting.

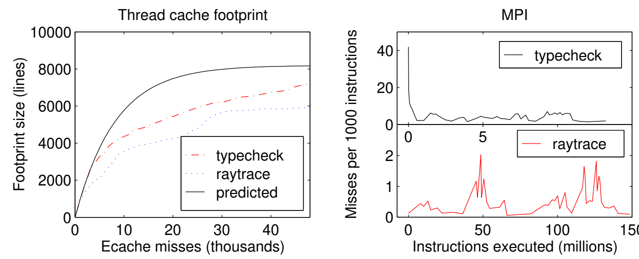

Figure 7: Overestimated footprints.

4 Scheduling Policies

The previous section described an analytical model for predicting thread footprints. We now turn to several practical scheduling policies that use this model to efficiently implement locality scheduling. To be effective, the scheduling overhead imposed by any such policy must be less than the avoided cache reload penalty. In this section we examine the data structures, priority schemes, and the general implementation ideas and techniques used to lower the rescheduling overhead. In particular, we establish several priority-based thread scheduling frameworks specifically designed to minimize the runtime overhead.

4.1 Largest Footprint First Policy (LFF)

The Largest Footprint First (LFF) policy is a simple greedy scheduling policy that, at each rescheduling point for a processor, dispatches the runnable thread with the largest footprint in the cache of that processor.

To be efficient, LFF must have fast access to the thread with the largest footprint in the cache of each processor. In general, any priority data structure can ensure the needed semantics. However, simply having fast access to the thread with the largest footprint is not sufficient. In addition, we need to be able to efficiently update the affected footprints during each context switch.

Consider a general situation: thread A executing on processor p blocks (or yields) at moment t. In Section 2, we have established the theoretical framework that allows us to compute the new footprints of all threads at the time of context switch t, given their previous footprints and the number of misses n taken by thread A during the scheduling interval. Since a thread running on a processor affects the footprints of all threads whose state is partially cached by that processor, conceptually, during a context switch, we need to recompute the footprints of ***all*** threads reflecting the activity of the blocking thread.

Active Threads, our thread package, can support millions of threads with a basic context switch cost on the order of 100 instructions on a variety of modern architectures [33]. Having to perform $O(T)$ operations to compute new footprints during a context switch (where T is a total number of threads), would not achieve any performance gains for fine-grained parallel applications with large T. In particular, it is desirable to keep the priority update overhead close to the basic context switch cost (saving and restoring registers, manipulating the thread context block data structure, etc.) and ensure that it is independent of the total number of threads T.

Our goal is to use the theoretical results from the previous sections to develop the data-structures and algorithms that would require only $O(d)$ operations during a context switch, where d is the number of dependencies of the blocking thread A (the out-degree of node A in graph G). In other words, we would like to avoid any computations for threads that do not depend on thread A in the dependency graph. To achieve that, we introduce a carefully designed priority function that is related to the thread footprint, but requires no updates in the common case of independent threads.

Let $m(t)$ be the total number of the secondary cache misses taken by processor p since the beginning of the *application* execution until time t. Let

$$k \equiv \left(\frac{N-1}{N}\right)$$

Instead of using the expected footprint size as thread priority, we use a related measure:

$$p_A(t) = \log\left(\frac{E[F_A]}{k^{m(t)}}\right)$$

It follows that at each point in time, for two threads A and B

$$(p_A(t) < p_B(t)) \Leftrightarrow (E[F_A] < E[F_B])$$

Therefore, our priority scheme works exactly as if we used the actual expected thread footprints for thread priorities. An interesting property of the chosen function is that it remains constant during the entire scheduling interval of thread A for all threads that are independent of A. Intuitively, this effect is achieved by carefully inflating the priorities of thread A itself and all dependent threads while making sure that the relationship between thread priorities and footprints is preserved. The computation of the expected footprint sizes can therefore be omitted in the common case of independent threads.

We consider how thread priorities need to be updated when thread A blocks. As before, we look at the three cases: 1) thread A itself, 2) threads dependent on A; and 3) threads independent of A.

Before we proceed, we need one more definition: let t_0 be the time when the blocking thread A was last dispatched on processor p (the beginning of the scheduling interval $[t_0, t]$). We then have $m(t)=m(t_0)+n$, where n is the number of misses during the interval.

Case 1. Blocking Thread A.

From Section 2, the new expected footprint of A is

$$E[F_A] = N - (N - S_A)k^n$$

Therefore, the new priority of the blocking thread at time t is

$$p_A(t) = \log\left(\frac{N-(N-S_A)k^n}{k^{m(t_0)+n}}\right) = \log(N-(N-S)k^n) - (m(t_0)+n)\log k$$

In the implementation of this policy, we pre-compute the values of k^n for a sufficiently large range of n (k asymptotically approaches 0). We also pre-compute all values of $log(F)$, $0 < F \le N$ (N, the total number of cache lines, is on the order of several thousands for most modern machines). The new priority of the blocking thread can then be computed in just few instructions: $logk$ is a constant, and n is reported by the performance counters.

Case 2. Dependent Threads

The new priority of the dependent thread C is given by:

$$p_C(t) = \log(qN-(qN-S)k^n) - (m(t_0)+n)\log k$$

As before, we rely on the statically pre-computed values of powers and logarithms (from case 1) to limit the cost of the priority update operation to just several instructions.

Case 3. Independent Threads

We now consider the independent thread B:

$$E[F_B] = S_B k^n$$

The new priority is given by:

$$p_B(t) = \log\left(\frac{S_B k^n}{k^{m(t_0)+n}}\right) = \log\left(\frac{S_B}{k^{m(t_0)}}\right) = p_B(t_0)$$

Thus, the priorities of the independent threads do not change, do not need to be updated at each context switch, and the common case incurs no runtime overhead. This is possible because our scheme intentionally inflates priorities in the other two cases.

We have identified a practical priority scheme that is equivalent to scheduling threads with the largest footprints (LFF), but requires no computations in the common case of independent threads. Priority updates for the blocking thread and dependent threads require only several floating point instructions per thread.

We next integrate a variant of a policy previously proposed in the context of independent tasks into our theoretical framework for shared thread state while also paying a low price for thread priority updates.

4.2 Smallest Cache-Reload Ratio (CRT)

Squillante and Lazowska suggested the use of the cache-reload miss ratio function to guide scheduling of independent tasks [21]. This policy assumes that each thread operates within a particular footprint and a thread dispatched after an inactive period must rebuild the footprint in the cache that existed when the thread blocked. This scheduling policy generally favors threads with the largest portion of their footprint still in the cache. A set of simulations demonstrated the performance potential of this policy in the context of independent tasks [21]. We extend this previous work to the case of unrestricted threads that can share text and data and design a priority scheme to allow an efficient implementation of CRT on real platforms (not just simulated memory systems).

For an inactive thread at a scheduling point, the cache-reload miss ratio can be expressed as

$$R(\dot{F}) = \frac{\dot{F} - E[F]}{\dot{F}}$$

where $\dot{F}$ is the total number of lines that a task will need in the future and $E[F]$ is the current expected footprint. In general, $\dot{F}$ is unknown and may vary from run to run. However, if we assume that the thread had all its state in the cache when it executed last, we will have a related ratio function:

$$R = \frac{E[F_0] - E[F]}{E[F_0]}$$

where $E[F_0]$ is the known expected footprint of a thread computed when it last executed on the considered processor.

As in the case of LFF, we would like to select a priority scheme such that the priorities of independent threads remain unchanged (and other priorities are inflated to make this possible). The priority function with the desired properties in this case is

$$p(t) = \log\left(\frac{1-R}{k^{m(t)}}\right)$$

where $m(t)$ is defined as previously to be the total number of cache misses for the considered processor from the beginning of *program* execution. Higher priorities mean lower cache-reload miss ratio: threads with the low reload miss ratio are better candidates for dispatch since they have a larger portion of state in the cache. For two threads A and B at any moment we have:

$$R_A < R_B \Leftrightarrow p_A(t) > p_B(t)$$

Once again, we turn to the now familiar three cases.

Case1: Blocking thread

For blocking thread A, we have $R = 0$ and

$$p_A(t) = -m(t)\log k$$

Since *logk* is a constant and the runtime accumulates *m*(*t*) anyhow, we need just two (or even one if *-logk* is pre-computed) floating point instructions to update priority in this case.

Case2: Dependent thread

Plugging in the derived expression for $E[F_C]$ in the priority formula, we obtain the new priority of the dependent thread C

$$p_C(t) = \log(E[F_C]) - \log(E[F_{C,0}]) - m(t)\log k$$

As before, we rely on the pre-computed values of *log*(*F*) for all $0 < F \le N$ to reduce the cost of priority update to just a few instructions.

Case3: Independent thread

Let t_0 be the point in time when independent thread B last executed on processor *p*. Then, the new priority of thread B is

$$p_B(t) = \log\left(\frac{1}{k^{m(t)}}\left(1 - \frac{E[F_{B,0}](1 - k^{(m(t) - m(t_0))})}{E[F_{B,0}]}\right)\right) = \log\left(\frac{1}{k^{m(t_0)}}\right)$$

Since $\log\left(\frac{1}{k^{m(t_0)}}\right) = p_B(t_0)$, after thread B blocks, its priority does not change and does not need to be updated. Just as in the case of LFF, the chosen priority scheme results in no computations for the common case of independent threads.

Table 3 summarizes the costs of priority updates per thread in terms of floating point instructions for LFF and CRT utilizing the described priority schemes.

Thread kind	LFF, flops	CRT, flops
blocking	6	2
independent	0	0
dependent	8	4

Table 3: The costs of priority updates

Both policies can be implemented extremely efficiently, with priority updates adding a relatively low additional overhead to the cost of maintaining priority data structures. For sufficiently sparse dependency graphs, the scheduling overhead of the two considered locality policies is dominated by the cost of maintaining priority data structures (heaps, etc.) and is comparable to the overhead of simple priority scheduling.

5 Implementation and Results

We have implemented the two scheduling policies described in the previous section in the framework of Active Threads. Active Threads is a high-performance portable threads system that supports blocking threads and a variety of synchronization objects: mutual exclusion locks, semaphores, barriers, condition variable, etc. The scheduling event mechanism of Active Threads is designed to support a variety of specialized scheduling polices [33]. The system runs on several hardware platform including SPARC, Intel 386 and higher, DEC Alpha AXP, HPPA, and MIPS and is integrated with the Sather compiler distribution.

Both implementations maintain the dependency graphs induced by user annotations and recompute the affected priorities during context switches. Both policies use the same binary heap data structure associated with each processor of the multiprocessor. As an optimization, threads whose footprints drop below a certain threshold on some heap, are removed from that heap to bound heap sizes and keep the cost of elementary heap operations low. If a thread is removed from all heaps, it is added to a single global queue. A processor whose heap becomes empty - which signifies that no existing threads have much state cached by this processor - consults the global queue for threads to dispatch. If the queue is also empty, an idle processor steals a thread with the lowest priority from a neighbor to balance load.

We compare the performance impact of the cache affinity scheduling policies for several fine-grained parallel applications and benchmarks: *tasks*, *merge*, *photo,* and *tsp* (the applications from the SPLASH suite that were used in our previous simulations do not exemplify the thread programming model: they are coarse-grained with the number of threads matching the number of processors; often explicitly tuned for locality and load balancing).

The base case for our experiments is a first-come first-serve policy (FCFS). The reported numbers were obtained with the help of the UltraSPARC performance instrumentation counters (PICs) and a high-resolution nanosecond wall clock timer.

Tasks is a benchmark previously used in the simulations to evaluate the potential benefits of using processor-cache affinity information in scheduling on SMPs [21]. *Tasks* creates a fixed number of identical threads with equal size, but disjoint footprints that repeatedly wake up, touch their state, and block for the same duration that they were active. Since tasks have disjoint states, user annotations are not relevant in this case.

Merge and *photo* were described previously in Section 3.3. For *merge*, user annotations reflect the fact that parent threads access the state prefetched by the children. In *photo*, a separate thread is created to retouch each row of pixels. During the course of computation, a thread accesses the states of several “neighbor” rows. The annotations indicate that the closer the corresponding row numbers, the more prefetched state is reused.

Tsp solves the Traveling Salesman Problem using the branch-and-bound algorithm: the solution space is repeatedly divided into two subspaces for the solutions with a given edge and those without the edge [20]. Solution subspaces are represented as adjacency matrices. Partial paths and several other auxiliary data structures are implemented by linked structures. The application is irregular in nature and performs a significant fraction of time accessing data.

The input parameters for the application runs are summarized in Table 4. We are mostly interested in the applications with fine-grained threads, and the footprint size in *tasks* was set to 100 lines. The other applications demonstrate fairly fine-grained parallelism as well. *tsp* is non-deterministic and different scheduling policies and even different runs for the same policy result in different execution paths. For this program, we have recorded the task sizes created for an LFF in a parallel run and benchmarked other policies and platforms for equal “work”.

Our experiments were conducted on two systems: a stand-alone UltraSPARC-1 and an 8 processor Sun Enterprise 5000 server.

tasks	1024 tasks, footprints 100 lines each, 100 scheduling periods per task.
merge	100,000 uniformly distributed elements. Switches to insertion sort for tasks of size 100 or smaller; creates 1024 threads.
photo	applies a "softening" filter to an rgb pixmap of size 2048x2048; creates 2048 threads.
tsp	finds a suboptimal path for the traveling salesman problem for 100 cities; measured the execution of 1000 threads.

Table 4: Input parameters for application runs.

Both platforms utilize 167Mhz UltraSPARC 1 processors whose memory hierarchy characteristics were used to drive the simulations of Section 3 and were given previously in Table 1. Both platforms run the Solaris 2.5.1 operating system. The single processor workstation has 128Mb of main memory and the Enterprise 5000 has a total of 512Mb of RAM. All runs fit within the available memory on either platform.

Processors of the Enterprise 5000 are connected via cache-coherent Sun Gigaplane Interconnect that delivers up to 2.7GB/s of bandwidth [22]. The E-cache (L2) miss latency on the UltraSPARC-1 is 42 cycles. On the Enterprise 5000, an E-cache miss requires 50 cycles, if the line is not cached by another processor, and 80 cycles otherwise.

On both platforms, the two 32bit performance counters (PICs) of all processors are configured to accumulate the number of E-cache references and hits. These counters are read at user-level during each context switch to compute the number of E-cache misses for the scheduling period. The counter overhead includes only several instructions for reading and resetting the appropriate registers.

Figure 8 shows performance on a single processing Ultra-1 workstation. Both policies achieve substantial improvements in terms of the number of E-cache misses and overall performance for *tasks* and *merge*. Both LFF and CRT versions of *tasks* run more than twice faster than the FCFS version.

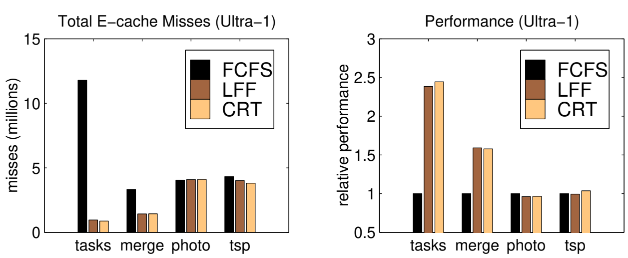

Figure 8: Performance impact of locality scheduling on a single processor Ultra-1.

For *tsp* on the Ultra-1, both policies eliminate only a moderate number of E-cache misses (up to 12% for CRT). When the solution space for *tsp* is split into two subspaces, the new objects for subspaces are allocated on the heap and initialized using the information about the original solution space. The misses associated with the initialization stage for each thread are compulsory and cannot be eliminated by any scheduling policy. However, parent threads prefetch some data for children which is reflected by the annotations.

In the case of *photo* on a single processor, the base FCFS policy happens to be very well suited for cache reuse. In fact, in our single processor traces, both locality guided policies result in a scheduling order close to FCFS. Since locality policies utilize

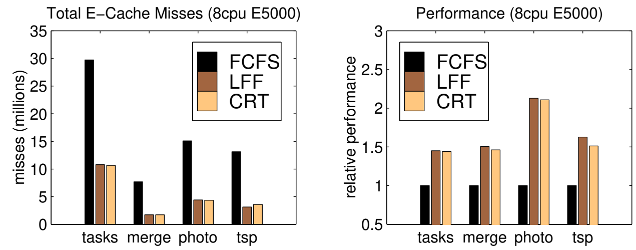

Figure 9: Performance impact of locality scheduling on the 8cpu Sun Enterprise 5000.

substantially more complex data structures, the resulting performance is somewhat less than that of the base FCFS case. This allows us to estimate the performance implications of the increased complexity of the underlying scheduling data structures. As Table 5 indicates, the policy that optimizes cache reload transient induces an extra 1% of E-cache misses and the resulting executable is about 3% slower than the base FCFS version on the 1cpu UltraSPARC-1 - a moderate price to pay for large potential improvements. For instance, on a multiprocessor, a simple FCFS policy is no longer optimal for *photo* and both locality-conscious policies eliminate about 70% of all E-cache misses improving the overall performance by over a factor of 2 (Figure 9).

Figure 9 presents the measurements obtained on the eight processor Sun Enterprise 5000. It shows that locality scheduling eliminates 60-80% of all E-cache misses for all considered applications. The overall performance is improved by factors of 1.45-2.12. Table 5 summarizes the performance implications of CRT vs. FCFS. Numbers for LFF are quite similar.

	E-misses eliminated%		Relative Performance	
	1cpu Ultra-1	8cpu E5000	1cpu Ultra-1	8cpu E5000
tasks	92%	64%	2.38	1.45
merge	57%	77%	1.59	1.50
photo	-1%	71%	0.97	2.12
tsp	12%	73%	1.04	1.51

Table 5: CRT relative to FCFS

While all considered applications demonstrate significant performance gains on the Sun E5000, for different applications, speedup comes from different sources. *tasks* generates long lived disjoint threads and in the absence of unnecessary annotations, speedup is achieved by preserving locality within each thread. Footprints are computed based on the cache performance feedback exclusively. *merge* achieves speedup almost entirely through user annotations: very light-weight threads are created to perform a single operation, but substantial locality across threads exists for any path in a task tree from the root to the leafs. While *tsp* has a somewhat similar task graph, its speedup is mostly due to preserving the locality within a thread. Threads in *tsp* are more persistent - they update global data structures, create and traverse new linked structures, etc. Global updates and memory allocation for new objects require synchronization (we are currently using a standard Solaris memory allocator protected by the mutual exclusion lock). Even in the absence of user annotations in the case of *tsp*, locality within a thread is preserved by using the footprint model computed from the cache use information. Adding annotations does not improve performance much further.

For *photo* both kinds of information appear to be critical. For instance, the LFF policy in the absence of annotations still eliminates 41% of all misses that are eliminated when the annotations are present. Similarly, in the absence of annotations, LFF achieves 53% of possible speedup. In general, the impact of annotations appears to be more profound for more light-weight and specialized tasks with a large degree of sharing between the tasks.

In our runs, the two locality policies demonstrate quite similar performance. Both LFF and CRT policies are greedy in the sense that the dispatched thread satisfies the local optimality criterion. The difference between the policies is the nature of this optimality criterion. Future experiments are necessary to identify the contexts in which one policy consistently outperforms the other.

6 Related Work

The problem of the cache interaction of the two independent tasks was initially studied by Thiébaut and Stone [31]. Their classical work also introduced the terminology used in this paper and inspired many later cache models.

The Analytical Cache Model by Agarwal et al. [2] provides the estimates of the cache performance based upon a number of parameters extracted from the address traces.

A recent work of Falsafi and Wood extends the Thiébaut and Stone cache model for disjoint address spaces to the case of limited sharing in the context of a performance model of the Wisconsin Wind Tunnel (WWT) [9]. In WWT, target processes have disjoint private data segments and a single fixed size shared text segment. The cache model is designed based on the assumption that all processes have the same shared footprint (text segment). To the best of our knowledge, this is the only prior attempt to integrate limited sharing of address spaces into a cache model. The derived model appears to be too expensive computationally for on-line use.

Squillante and Lazowska introduced the cache reload miss ratio function and performed simulations to investigate the potential of different locality policies [21]. They also mentioned a number of implementation techniques that influenced this work and were the first to argue for the possible use of cache performance counters for thread scheduling.

A system that uses hardware feedback to schedule threads on a NUMA architecture is described by Bellosa and Steckermaier in [3]. To our knowledge, this is the first *working* system (and the only one, prior to Active Threads) that attempts to use the on-line performance feedback for thread scheduling. The current work has also been influenced by the analytical techniques of [3]. The conducted experiments were somewhat limited by the Convex SPP 1000 hardware deficiencies: processors cannot distinguish between processor and network-cache misses and the cost of reading cache miss counters is equivalent to a user-level context switch.

The memory-conscious scheduling policy of [15] suggests a combination of a static initial mapping for locality with dynamic load balancing to improve performance of fine-grained threads.

Several systems are based exclusively on user annotations to improve the performance of threaded applications. [17] uses address hints at thread creation time to implement runtime tiling of data through the use of run-to-completion threads on serial machines. [25] investigates the high-level object-oriented constructs to express locality in a portable way.

Several OS kernel paging techniques have been proposed to improve data locality. [5] suggests the use of VM to avoid hot spots in large direct-mapped secondary caches dynamically. [13] argues for careful initial VM mapping to reduce conflict misses.

7 Conclusions and Future Work

This paper describes a combined approach for improving locality that uses the hardware performance monitors of modern processors and program-centric code annotations to guide thread scheduling on SMPs. The approach demonstrates a substantial potential for reducing the number of secondary cache misses and improving the overall performance.

An important open question is whether the current annotations support a simple enough programming style to be used in large and complex systems. It is currently required that important sharing information be identified by the user and the system does not attempt to infer any relationships. However, such specifications may be too tedious for large systems. It remains to be seen how much dependency propagation and inference can be off-loaded onto the runtime system and static compiler analysis while keeping the performance benefits of the basic model described here.

It is even more attractive to identify state sharing patterns entirely at runtime to handle, for instance, the existing unmodified POSIX and Java Threads application bases. Bershad et al. suggested the use of a Cache Miss Lookaside buffer (CML), an inexpensive hardware device placed between the cache and main memory, to detect conflicts by recording a miss history at a page granularity [5]. While the problem of identifying sharing rather then conflicts appears to be more difficult (large cache miss penalties provide an ample opportunity for CML bookkeeping of misses), perhaps with the use of a related hardware device combined with the VM techniques, some sharing patterns could be inferred without user intervention. Repeated trial runs with judicial unmapping of pages at the context switch time may be another viable alternative for identifying shared pages.

Rearranging the execution order may have an adverse effect on fairness. In particular, locality techniques generally favor the execution of a few threads with much state already in the cache possibly starving the others. While fairness is not an issue in our domain (all threads must run to completion and constraints are specified explicitly through synchronization), if fairness is important, a practical scheduler must provide an escape mechanism to bypass the default priority evaluation.

8 Acknowledgments

We would like to thank Ben Gomes, Jerry Feldman, Hans-Arno Jacobsen, Chu Cheow Lim and the anonymous reviewers for their valuable comments on earlier versions of this paper. We also thank Eric Fraser, Steve Lumetta and the NOW group of UC Berkeley for providing the hardware infrastructure and technical assistance for this research.

References

[1] T. E. Anderson, E. D. Lazowska, H. M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Comput.* 38, 12, December 1992, 1631-1644. Also appeared in *ACM SIGMETRICS*, May 1989.

[2] A. Agarwal, M. Horowitz, J. Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, Vol. 7, No.

2, May 1989, pp. 184-215.

[3] F. Bellosa and M. Steckermaier. The Performance Implications of Locality Information Usage in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing* 37, 1996. pp. 113-121.

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

[5] B. N. Bershad, J. B. Chen, D. Lee, T. H. Romer. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, Ca. 1994. pp. 158-170.

[6] 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.

[7] R. F. Cmelick and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. TR UWCSE 93-06-06, Department of Computer Science and Engineering, University of Washington.

[8] Z. Cvetanovic, D. Donaldson. AlphaServer 4100 Performance Characterization. *Digital Technical Journal*, April 1997.

[9] B. Falsafi, D. Wood, Modeling Cost/Performance of a Parallel Computer Simulator, *ACM Transactions on Modeling and Computer Simulation*, 1997.

[10] Hewlett Packard. HP Exemplar Technical Servers, Technical Specification, available at http://www.hp.com/wsg/products/servers/servhome.html

[11] 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

[12] Intel Corporation. Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Guide, December 1995.

[13] R. E. Kessler and M. K. Hill. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Transactions on Computer Systems*, Vol. 10, No. 4, November 1992, pp. 338-359.

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

[15] 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 Trends in Parallel and Distributed Computing. World Scientific Publishing, 1994.

[16] L. McVoy and C. Staelin. Imbench: Portable Tools for Performance Analysis, *USENIX 1996 Annual Technical Conference* (1996), pp. 279-294.

[17] J. Philbin and J. Edler. Thread Scheduling for Cache Locality. *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996, Cambridge, Massachusetts, US. pp. 60-71.

[18] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Ketone, C. Kozyrakis, R. Thomas, K. Yelick. Intelligent RAM (IRAM): Chips that remember and compute. *1997 IEEE International Solid-State Circuits Conference*. San Francisco, Ca, 6-8 February, 1997.

[19] J. W. Quittek and B. Weissman. Efficient Extensible Synchronization in Sather. *Scientific Computing in Object-Oriented Parallel Environments. First International Conference (ISCOPE 97)*, December 1997. LNCS 1343, Springer-Verlag, 1997, pp. 65-72.

[20] E. M. Reingold, J. Nievergelt, N. Deo. Combinatorial Algorithms: Theory and Practice. Prentice-Hall, 1977.

[21] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 2, February 1993. pp. 131-143.

[22] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, B. Lienscres. Gigaplane: A High Performance Bus for Large SMPs, *Hot Interconnects IV*, Stanford, Ca, 1996, pp. 41-52.

[23] M. B. Steinman, G. J. Harris, A. Kocev, V. C. Lamere, and R. D. Pannell. The AlphaServer 4100 Cached Processor Module Architecture and Design. *Digital Technical Journal*, April 1997.

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

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

[26] Sun Microsystems Lab. Shade User's Manual, V.5.32C, 1997.

[27] Sun Microsystems Inc. UltraSPARC-1 User's Manual, 1996.

[28] Sun Microsystems. The UltraSPARC Processor Technology White Paper: http://www.sun.com/microelectronics/whitepapers/UltraSPARCtechnology

[29] Sun Microsystems. UltraSPARC-1 Data Sheet. First Generation SPARC v9 64-Bit Microprocessor. STP1030A, Sun Microelectronics, July 1997.

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

[31] D. Thiébaut, H. S. Stone. Footprints in the Cache. *ACM Transactions on Computer Systems*, Vol. 5. No. 4, November 1987, pp. 305-329.

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

[33] B. Weissman and B. Gomes. Active Threads: Enabling Fine-Grained Parallelism in Object-Oriented Languages. *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 98)*, July 1998.

[34] E. H. Welbon, C. C. Chan-Nui, D. J. Shippy, D. A. Hicks. The POWER2 Performance Monitor. IBM internal paper.

[35] 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.

Appendix

This appendix derives the expected footprint size of dependent thread C after n misses by thread A (Section 2.4).

We model the cache behavior of thread C with a Markov chain of Figure 10. Each state in the Markov chain corresponds to the number of lines of thread C present in the cache of processor p. Each edge represents a transition for C's footprint caused by a miss

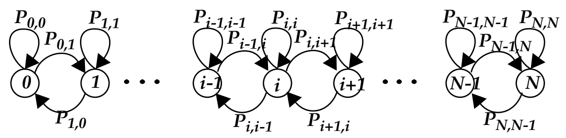

Figure 10: A Markov chain simulating the cache transitions of dependent thread C.

taken by A and is labeled with the transition probability. There is a total of N states, although depending on the number of misses n and the initial state S_C, some of the states in the chain may be unreachable during a particular scheduling interval.

Suppose, at some time τ, thread C that shares some of its state with A has i lines in the cache of processor p. At this time, thread A takes a miss on processor p. This miss can trigger any of the following transitions for the cached state of thread C:

- increase C's footprint if a new line of data shared between A and C is brought into the cache and it does not overwrite any of the state of C already in the cache
- decrease C's footprint if a new line contains data not shared with C and a line with C data is evicted to accommodate the new line
- no change to C's footprint. This is possible in two cases:
  a) the new line is shared with C, but overwrites another C line;
  b) the new line is not shared with C and it does not overwrite any C line.

In Figure 10, these three possibilities correspond to transitions from the state i to states $i+1$, $i-1$, and back to i respectively.

We now derive the probabilities of these transitions. Since in state i, i lines of C are present in the cache, the probability of this number going up by one on a cache miss by thread A is

$$P_{i,\,i+1} = q_{A,\,C}\left(\frac{N-i}{N}\right)$$

where $q_{A,C}$ is a sharing coefficient between threads A and C (a weight of $(A,\ C)$ in the dependency graph G). The probability of the number of C lines decreasing by one is

$$P_{i,\,i-1} = (1-q_{A,\,C})\frac{i}{N}$$

Finally, the probability of the C's footprint size remaining unchanged consists of two parts. The probability that the new line contains shared data and it overwrites another C line is $q_{A,\,C}\frac{i}{N}$. The probability that it does not contain any shared data and it does not cause eviction of any C lines is $(1-q_{A,\,C})\left(\frac{N-i}{N}\right)$. Therefore,

$$P_{i,\,i} = q_{A,\,C}\frac{i}{N} + (1-q_{A,\,C})\left(\frac{N-i}{N}\right)$$

The generator matrix for our Markov chain is the following matrix ***M***:

$$\begin{matrix}
1-q & q & 0 & 0 & \dots & 0 \\
(1-q)\frac{1}{N} & \frac{q}{N}+(1-q)\left(\frac{N-1}{N}\right) & q\left(\frac{N-1}{N}\right) & 0 & \dots & 0 \\
0 & (1-q)\frac{2}{N} & q\frac{2}{N}+(1-q)\left(\frac{N-2}{N}\right) & q\left(\frac{N-2}{N}\right) & \dots & 0 \\
\dots & \dots & \dots & \dots & \dots & \dots \\
0 & 0 & \dots & (1-q)\left(\frac{N-1}{N}\right) & q\left(\frac{N-1}{N}\right)+\frac{(1-q)}{N} & q\frac{1}{N} \\
0 & 0 & \dots & 0 & 1-q & q
\end{matrix}$$

We designate vectors and matrices by boldface identifiers with no subscripts. Element $m_{i,j}$ of ***M*** is the probability that a single cache miss for thread A will change the number of cache lines that belong to thread C from i to j. Therefore, ***M*** is a tri-diagonal matrix. Raising ***M*** to power n yields matrix $\boldsymbol{M^n}$ such that element $m^n_{i,\,j}$ is the probability of the transition from state i to j after n misses by thread A.

Given matrix $\boldsymbol{M^n}$, we can compute the expected size of C's footprint after n misses by thread A, assuming that the initial number of such lines is S_C:

$$E^n_{S_C}[F_C] = \sum_{i=0}^{N} m^n_{S_C,\,i}\,i$$

It is convenient to generalize the above to the vector of all expectations indexed by the initial footprint size. A vector element with the index corresponding to a certain initial footprint represents the expected footprint value after n cache misses:

$$\boldsymbol{E^n}[F_C] = \boldsymbol{MM}\dots\boldsymbol{MT^0}$$

where $\boldsymbol{T^0} = [0, 1, 2, \dots N]^T$.

To simplify the derivations, instead of considering matrix/matrix products, we will move from right to left in the above expression for $\boldsymbol{E^n}[F_C]$evaluating a series of vector/matrix products. Consider the first matrix/vector product $\boldsymbol{MT^0}$.

$$\boldsymbol{T^1} = \boldsymbol{MT^0} = \boldsymbol{T^0}\left(\frac{N-1}{N}\right) + \boldsymbol{e}q_{A,\,C}$$

where $\boldsymbol{e}$ is a vector of all ones of the same size as $\boldsymbol{T^0}$.

In fact, by noting that a vector of all ones completely absorbs ***M*** ($\boldsymbol{Me} = \boldsymbol{e}$), we obtain

$$\boldsymbol{T^{i+1}} = \boldsymbol{MT^i} = \boldsymbol{T^i}\left(\frac{N-1}{N}\right) + \boldsymbol{e}q_{A,\,C}$$

We now have a simple recurrence that leads to the following expression for $\boldsymbol{T^n}$:

$$\boldsymbol{T^n} = \boldsymbol{e}q_{A,\,C}\sum_{i=0}^{n-1}\left(\frac{N-1}{N}\right)^i + \boldsymbol{T^0}\left(\frac{N-1}{N}\right)^n$$

Simplifying the above (by noting a geometric sum), we obtain a closed-form expression:

$$\boldsymbol{T^n} = \boldsymbol{e}q_{A,\,C}N - (\boldsymbol{e}q_{A,\,C}N - \boldsymbol{T^0})\left(\frac{N-1}{N}\right)^n$$

Since $\boldsymbol{E^n}[F_C] = \boldsymbol{T^n}$, the vector of all n-th order expectations of C's footprints (for all initial footprints) is:

$$\boldsymbol{E^n}[F_C] = \boldsymbol{e}q_{A,\,C}N - (\boldsymbol{e}q_{A,\,C}N - \boldsymbol{T^0})\left(\frac{N-1}{N}\right)^n$$

Finally, for a specific initial footprint S_C (since $\boldsymbol{T^0} = [0, 1, 2, \dots N]^T$, $T^0_{S_C} = S_C$), we have a scalar expression

$$E^n_{S_C}[F_C] = q_{A,\,C}N - (q_{A,\,C}N - S_C)\left(\frac{N-1}{N}\right)^n$$

If we substitute $q_{A,\,C} = 1$, i.e. complete overlap, into this closed-form solution for the expected number of lines for thread C, we obtain a previously derived solution for case 1). Alternatively, substituting $q_{A,\,C} = 0$ (threads have no shared data), will yield a solution for case 2) of Section 2.4.