Footprint-based Scheduling

Marc de Kruijf and Gwendolyn Stockman

CS736 Course Project, Fall 2007
Department of Computer Sciences, University of Wisconsin–Madison

Abstract

As we move into the multicore era, fine-grain multithreading will be among the primary tools used by software developers to enhance the parallel performance of their applications. However, more so than in prior, coarse-grain multithreading systems, intelligent locality-aware scheduling of fine-grain threads is paramount for good performance. Fortunately, in fine-grain multithreaded applications there is an abundance of threads eligible for execution at any given time, suggesting opportunities to develop efficient, locality-aware thread scheduling algorithms that can improve overall performance.

In this paper, we develop techniques for footprint-based thread scheduling – a previously uncharted area of research. We experimentally evaluate two sets of synthetic workloads to demonstrate the importance of locality-aware scheduling. For array-based workloads, we show a worst-case 5X performance difference between locality-aware and locality-indifferent executions. Similarly, for tree-based workloads, we show in excess of a 2X performance difference across best-case and worst-case executions. We also discuss and present architectural enhancements to expose thread footprint information to software for efficient footprint-based thread scheduling. Finally, we describe a footprint-based thread scheduling algorithm to be evaluated in future work.

1 Introduction

Multithreaded applications are becoming pervasive due to the emergence of multicore processors. Historically, multithreading has been used primarily to extract concurrency between I/O and computation, but it can also be used to enable concurrent computation on multiprocessor systems. One of the main problems with threads, however, is that their memory access behavior is completely invisible, which makes it challenging to schedule threads for optimal cache utilization and performance. In this paper, we address this problem by proposing thread scheduling based on cache footprint analysis.

There are effectively two varieties of multithreading: coarse-grain and fine-grain. For applications with coarse-grain multithreading, cache locality is not a pressing concern. These applications consist of relatively few threads with infrequent communication, and the threads are typically scheduled to run for an OS time slice, which is on the order of millions of processor cycles. At the end of a time slice, a "context switch" occurs and threads are rescheduled, but for coarse-grain threads the cost of re-establishing context by re-populating caches is amortized across the lifetime of the thread's execution, and is therefore relatively negligible. Furthermore, these coarse-grain threads have only gotten relatively longer with time, as processor speeds have radically improved and demands on system interactivity (in terms of the OS time slice) have remained relatively constant. Hence, there has historically not been significant incentive to research this topic.

In contrast to coarse-grain multithreading, fine-grain multithreading involves of an abundance of threads with frequent communication and short execution times, typically only 100 to 10,000 cycles. Applications with fine-grain multithreading have many frequently executed, independent regions of code that can be extracted for parallel execution. This style of multithreading has recently

gained attention for two reasons: (1) due to their short execution time, it is only suitable when the overheads of scheduling and communication are small, as is the case on multicore systems; and (2) due to technology trends, continued application performance growth must come primarily from explicit application-level parallelism, which is necessary to exploit innovations in multicore processor architectures. Fine-grain multithreading (sometimes called *task-level parallelism* in the literature) has the potential to significantly enhance parallel performance for a significant number of applications.

Despite its growing popularity, however, fine-grain multithreading has the central drawback that it generally achieves very poor cache re-use; threads have a relatively short duration and are context switched frequently without opportunity to leverage the data already in the cache. Hence, the processor must frequently stall to re-populate thread context, and the wait is often substantial as illustrated in Figure 1.

Yet there is often an abundance of fine-grain threads eligible for execution at any given time, and hence there is opportunity to develop efficient, locality-aware thread scheduling algorithms that can improve overall performance. In this paper, we build on this observation to develop techniques for *footprint-based* thread scheduling – a previously uncharted field of research.

The central contributions of this paper are as follows:

- An evaluation of two synthetic workloads – an array-based workload and a tree-based workload – on a uniprocessor system. This evaluation demonstrates the potential for performance improvement using locality-aware schedulers for fine-grain multithreaded applications.
- Proposed primitives for exposing hardware cache footprint information to a software-level scheduler environment. These primitives enable sophisticated, runtime scheduler adaptation with minimal overhead. To the best of our knowledge, this is the first effort exploring this area.
- A footprint-based uniprocessor scheduler algorithm to be evaluated in future work.

The remainder of this paper is organized as follows. Section 2 describes related work. In section 3, we describe our experimental methodology. Section 4 presents our results. In section 5, we discuss our results and present directions for future work. Finally, section 6 concludes.

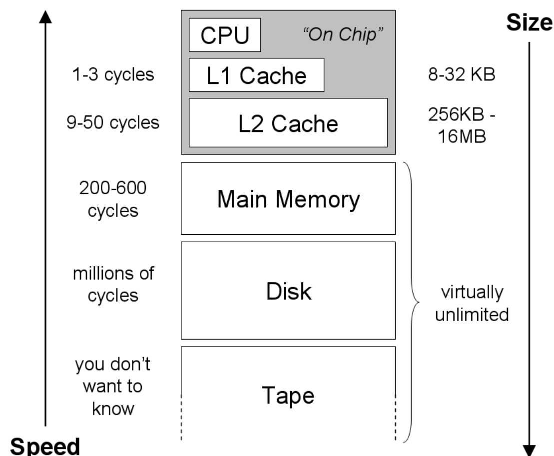


Figure 1: The memory hierarchy of a typical uniprocessor system.

2 Related Work

In this section, we open with related work in the area of multiprocessor scheduling, and end with more general work in the area of batched thread scheduling and other domain-specific locality-aware scheduling work.

2.1 Multiprocessor Thread Scheduling

While our work focuses primarily on uniprocessor systems, prior research in locality-aware scheduling has been directed primarily at multiprocessor systems. In these systems, there is an interesting trade-off between load balance and locality, and this trade-off has received significant attention in the literature.

Scheduling strategies for multiprocessor systems generally fall into one of three categories: *static scheduling*, *scheduling using shared queues*, and *scheduling using per-processor queues*.

In static scheduling [10], threads are mapped to processors as soon as they are created. Although this approach can result in locality-optimal execution, it compromises load balance and fairness, which can lead to poor performance.

Shared queues are the approach adopted in coarser-grain multithreading applications because they are simple and they effectively implement load balancing: when a processor becomes idle, it dynamically retrieves a thread from the shared queue. In this manner, threads are fairly scheduled and load is perpetually balanced across the available processors. This approach is widely used in coarse-grain threading environments where load balance is important and locality is not a significant part of the overall performance equation. It is also useful in fine-grain multithreaded applications where there is simply no data or code sharing between threads. An implementation of MapReduce for the Cell processor is one example of an application that uses a shared queue in a fine-grain threading environment [4]; in MapReduce, there is no locality internal to the Map, Reduce, and grouping phases.

The shared queue model has two primary disadvantages, however: first, access to a shared queue results in frequent contention for shared resources, as observed by Anderson *et al.* [1]; and second, cache utilization is poor since locality is not preserved across threads that share data. To manage these problems, fine-grain thread schedulers have traditionally used per-processor queues. Per-processor queues were favored by Eager *et al.*after they experimentally measured the costs associated with process migration [6]. Similarly, Squillante and Lazowska explored the locality benefits of establishing process affinity to a particular processor using queueing network models [13], and also quantified the trade-off between load balance and cache performance using their analytical model. Finally, Torellas *et al.*explored the benefits of affinity-based scheduling of multiprocessor workloads using per-processor queues, with analysis supported by low-level performance data [15].

There have also been several parallel language runtimes proposed using per-processor queues. Filaments [7] and Cilk [3] are two examples. Of particular interest, Cilk uses non-blocking double-ended deques for queues. Processors access their local queues by pushing and popping from the bottom, which maximizes locality. Remote processors access the queues by popping from the top. Results show that this strategy achieves very high efficiency due to strong locality preservation and minimal data structure contention. Lastly, Carbon [9] implements hardware support for Cilk-style per-processor queues, showing substantial performance improvement over a variety of software-based fine-grain thread schedulers.

2.2 Batched Thread Scheduling

Debattista *et al.*develop user-level thread scheduling algorithms that group fine-grain threads together into coarser-grain entities termed *batches*, allowing improved cache utilization on uniprocessors [5]. On shared memory processors, it further lowered contention for shared data structures and decreased the occurrence of *false sharing* (two threads on separate processors accessing different addresses that map to the same cache line), with only negligible impact to load balance.

Their basic technique is to use per-processor batch queues. When a thread spawns a new thread, that thread is added to the currently executing batch. New batches are created when the current batch overflows. This approach is very simple, and the rationale is that threads typically scheduled sporadically and subsequently suffering from the cache interference of intervening threads (as characterized by Thiebaut and Stone [14]) are scheduled several times within the same batch, exhibiting much better locality. Similarly, communicating threads tend to be co-located within the same batch, and are therefore able to make rapid forward progress.

SEDA [16] and StagedServer [11] are two event-driven architectures for scalable servers that process events in *stages*, using an approach similar to Debattista *et al.* to exploit locality. Staged processing separates out logical units of computation into stages and attempts to maximize processor cache locality by aggregating (batching) the execution of multiple similar events within an event queue, resulting in improved performance. SEDA also provides support for load balancing using re-assignment performed by dynamic resource controllers. StagedDB [8] applies this same idea to database request processing, breaking a request's execution into stages and processing a group of sub-requests at each stage.

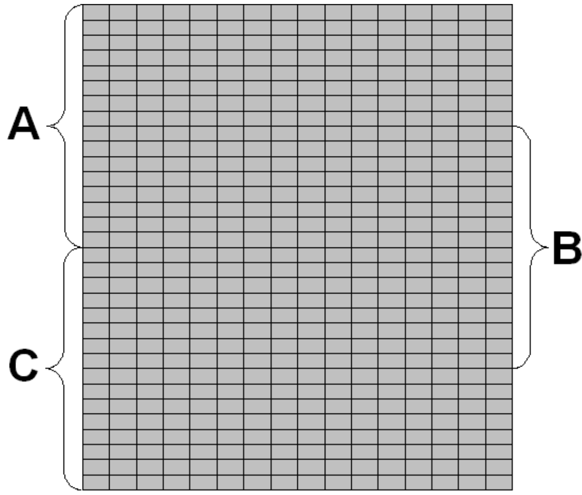


Figure 2: The portion of the array workloads A, B, and C touch.

```
for (r = low; r <= high; r++) {
  sum += array[r][0];
}
```

Figure 3: Pseudocode for array workloads A, B, and C.

3 Methodology

To examine the impact of locality on thread scheduling we examined two groups of synthetic workloads. The first group was a set of array-based workloads to help us understand the importance of locality for threaded applications with *regular* memory access patterns. The second group was a set of binary search tree (BST) traversal workloads to examine the locality behavior of threaded applications with *irregular* memory access patterns.

Our workloads were run using the GEMS simulation toolset [12]. Using the simulator allowed us to evaluate custom system configurations, such as direct-mapped L1 and L2 caches. The system configuration we used for our evaluation is shown in Table 1. To evaluate performance we divided the number of processor cycles by the number of instructions executed to derive the cycles per instruction (CPI). CPI is our performance metric, where a lower CPI indicates better performance.

3.1 Array-based Workloads

For the array-based workloads we used an integer (4 bytes) array of 512 rows by 16 columns, yielding a total size of 32 KB with one row being the size of one cache block. Figures 2 and 3 illustrate our three workloads on the array:

A: Access the first half of the array (rows 0-255)

B: Access the middle half of the array (rows 128-383)

C: Access the last half of the array (rows 256-511)

Several combinations of workloads were examined for cache locality between workloads. These include:

Combination 1: N occurrences each of workloads A and B.

Combination 2: N occurrences each of workloads A and C.

Combination 3: N occurrences each of workloads A, B, and C.

For each of these combinations two schedules were considered:

Schedule 1: Run all N instances of one workload consecutively, followed by N instances of the next workload, and so on.

Schedule 2: Run one instance of each workload, and then repeat N times in the same order.

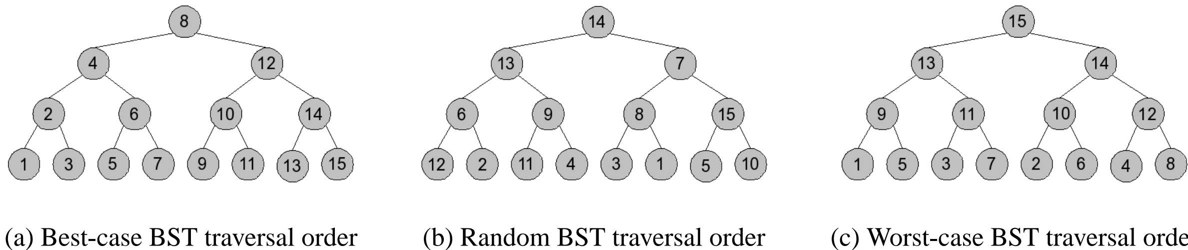
For example, if $N = 2$, then for Combination 1 we would have schedules: AABB and ACAC.

3.2 Tree-based Workloads

We used a BST of $2^{18} - 1$ nodes, where each node contained a word. Our workloads traverse the BST by searching for each word that comprises the BST in turn, where each search is conceptually a thread of execution that could potentially be scheduled. The different traversals are shown in Figure 4. The best-case traversal is an in-order traversal, where we first traverse the left subtree, then visit the root, and finally move to the right subtree. This is illustrated in Figure 4(a). The worst order to traverse the BST is depicted in Figure 4(c), where temporal

Component	Configuration
Processor	Sparc, dual-issue
L1 Cache	16 KB, direct-mapped, 1 cycle access latency, 64 byte block
L2 Cache	1 MB, direct-mapped, 13 cycle access latency, 64 byte block
Main Memory	1 GB, 200 cycle access latency

Table 1: GEMS simulator system configuration.



(a) Best-case BST traversal order (b) Random BST traversal order (c) Worst-case BST traversal order

Figure 4: Illustrations showing scaled-down versions of our three BST traversal workloads. The actual binary search tree had $2^{18} - 1$ nodes.

search path groupings are as dissimilar as possible. For a more general traversal case, we also evaluated a random tree traversal, where the order in which the words are searched is random, as shown in Figure 4(b).

4 Experimental Results

Here we present our experimental results for the different workload combinations and schedules discussed previously.

4.1 Array-based Workloads

We ran each workload combination described in section 3.1 with the number of occurrences of each workload (i.e. N) being 1, 10, 100, and 1000. We found $N = 1000$ to be the most accurate and thus will mainly present results for $N = 1000$. Also, we initialize the L1 and L2 caches before each experiment by iterating through the entire array from row 0 to 511, which results in all rows being in the L2 cache, and rows 256 to 511 being in the L1 cache at the start of each experiment. Since A is always the first workload run, it must always go to the L2 cache for each piece of data.

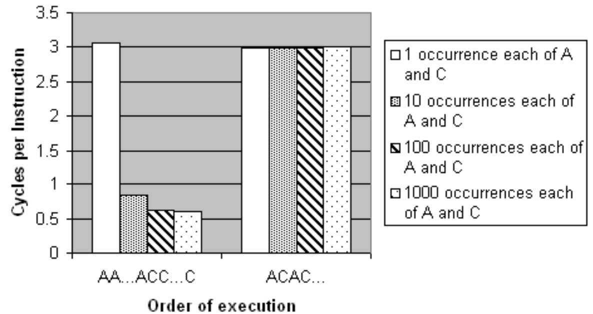


Figure 5: Results of Schedule 1 and Schedule 2 for a workload combination of A and C both run with 4 different values of N.

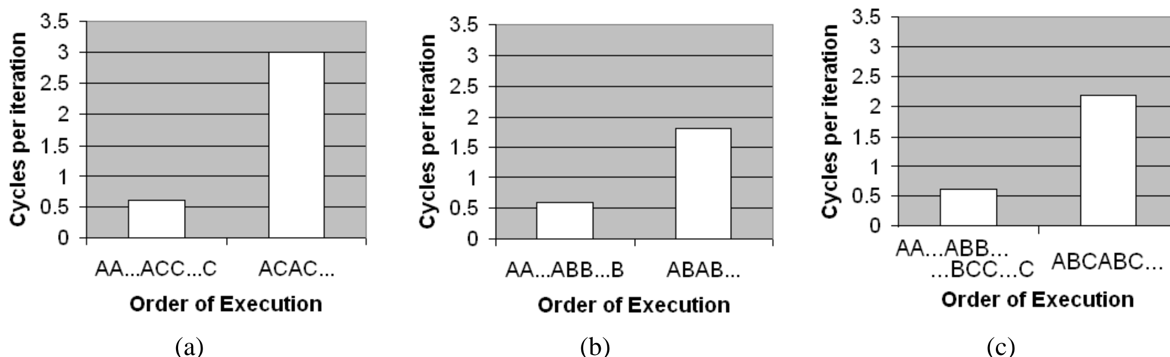


Figure 6: (a) Workload Combination of A and C. (b) Workload Combination of A and B. (c) Workload Combination of A, B, and C

First we will look at a workload combination of A and C. As you can see in Figure 5(a) when $N = 1$ the cycles per iteration (CPI) is about the same no matter which schedule is used. This is because none of the data in the L1 cache is used by the first the workload (A), and therefore it will always bypass the L1 cache and go to the L2 cache. However, as we increase N the CPI for Schedule 1 (run all the occurrences of A followed by all the occurrences of C) decreases, while the CPI for Schedule 2 (alternate between running A and C) remains the same. This relationship with N is similar for the other combinations and schedules examined, and the change in CPI between running each workload for 100 iterations and 1000 iterations is negligible. Thus, we shall only show results for $N = 1000$ for the other workload combinations.

Figure 6(b) shows us how the two different schedules perform with a workload combination of A and B. A comparison of Figure 6(b) and Figure 6(c) clearly shows the benefit of running workloads with cache footprints which partially overlap one after another. This also checks with what we expected–that running a combination of workloads A and B would perform better when following Schedule 2 then a combination of A and C. Finally, if we have a workload consisting of 1000 occurrences of each of A, B, C, we again see the benefit of overlapping cache footprints. A quick comparison of Figure 6(a) and Figure 6(c) show that there is a benefit to placing B between the calls of A and C. However, note that the performance of schedule 2 in Figure 6(c) is worse than the performance in Figure 6(b), that is because there is only a B placed between calls of A followed by C, not between calls of C followed by A, thus cache footprints are not utilized to their full potential in this algorithm.

Figure 7: Comparison of different BST traversals.

4.2 BST-based Workloads

The BST-based workloads are meant to be more complex than the previous array-based workloads. The BST used contained $2^{18} - 1$ nodes. Each node contained 1 word of about 11 bytes, and two 4 byte pointers, resulting in a total workload size of 4-5MB. The tree is so large that it is unlikely that for any two random nodes the traversal to them will follow many of the same branches, and as a result the two searches are likely to not interact well in the cache. This can be seen by comparing the performance

of the best-case workload to the randomized workload in Figure 7. In fact the randomized workload has performance significantly closer to that of the worst-case workload than that of the best-case workload. This is an example of how much cache-aware scheduling algorithms can help improve the performance of even randomized workloads, and how much is lost by not taking cache locality into account when deciding what order to schedule threads in.

5 Discussion and Future Work

Our experimental results indicate that there is potential for significant performance improvement using footprint-aware scheduling. However, cache footprint information cannot be readily extracted on current systems. At best, processors may provide performance counters to track cache hit and miss information over some interval of time. This information could be useful, but it does not expose sufficient memory context to draw accurate conclusions with respect to thread interactions. Similar information can be obtained using gray-box techniques [2], measuring the execution time relative to some dynamic instruction count and subsequently inferring hit and miss characteristics, but this approach is less flexible since it assumes that memory behavior is the primary source of variability in execution time, and is not appropriate if thread synchronization, for example, also contributes substantial variability. Furthermore, the problem of extraneous memory context remains.

The context of one or more thread executions can be isolated by loading data into the cache or flushing the cache of all relevant data in advance. This way, the execution is isolated from contextual effects, and thread interactions can be accurately measured. This technique is viable for static workloads, where a static schedule can be generated in advance and re-used across executions. However, for dynamic workloads, the cost of periodically overwriting or flushing the cache is likely to be prohibitive, contributing unacceptable overhead.

To resolve the problem of extraneous context, we propose a minor architectural extension to enable efficient dynamic footprint-based scheduling. In this section, we describe our proposal and the rationale behind it. We also describe a scheduling algorithm that harnesses this information, to be evaluated in future work.

5.1 Architectural Support for Footprint-based Scheduling

The opposite extreme of merely providing hit and miss counters is providing access to the entire cache state. However, from both a software and hardware perspective, the overhead of doing this is unreasonable; the required hardware support would be monstrous, and the software analysis required to extract meaningful scheduler data would be excessive. Rather, the hit and miss counts provide good starting points because the overhead is minimal and the information is already quite useful.

The enhancement we propose is to track hits and misses *relative only to the most recently executed thread* as follows. The hardware records the footprint of the most recent thread by tracking each cache line with a ”Touched” bit. Hits and misses are tracked for the current thread as normal, but *only the first hit or miss is recorded*. The reason is that only the first access to a given cache line matters. Subsequent accesses are internal to the thread and do not affect inter-thread scheduling choices. The sum of the hit and miss counts will be the total number of cache lines touched.

Table 2 shows the information tracked by hardware (bold) and the information to be exposed to software (italics): *Shared Hit* represents the number of cache lines hit on first access by the current thread and touched by the previous thread; *Shared Miss* represents the number of cache lines missed on first access by the current thread and touched by the previous thread; *Overwritten* represents the number of cache lines untouched by the current thread and also untouched by the previous thread; and *Unchanged* represents the number of cache lines touched by this thread and untouched by the previous thread.

Even though *Overwritten* and *Unchanged* both imply the absence of any interaction between the two threads for that particular cache line, we still choose to keep them distinct because the difference can be leveraged for more informed scheduling. For example, let us assume that thread t_2 executes after thread t_1, and that there are many lines *Unchanged* in this sequence. If we know some thread t_3 that has good sharing with t_1, we can schedule it subsequent to t_2 and be confident that much of the sharing will

	Current Thread Footprint			
Previous Thread Footprint		**Hit**	**Miss**	**Untouched**
	Touched	*Shared Hit*	*Shared Miss*	*Unchanged*
	Untouched	*Overwritten*		

Table 2: Hardware statistics exposed to software, summed over all cache lines: *Shared Hit*, *Shared Miss*, *Overwritten*, and *Unchanged*.

be preserved. In contrast, if many lines were *Overwritten*, it would not be a good idea, because the shared data will most likely have been evicted from the cache. This distinction can thus be useful from a scheduling perspective and introduces very little extra overhead.

The hardware support to enable this is a single extra bit per cache line to record whether that line was touched by the most recent thread, circuitry to populate and extract this data when a context switch occurs, and hardware interfaces to extract this information. We have implemented these changes as part of the GEMS simulation toolset [12]. The modifications have been tested and have been found to work successfully.

5.2 A Footprint-based Scheduling Algorithm

In this section, we present a simple footprint-based scheduling algorithm. First, we address which threads to consider when scheduling, and then we address how to schedule those threads. Suppose we have a stream of threads to run. If it is a finite stream, we could go through all permutations in order to find the best ordering with respect to cache locality, however this may take too long and assuming a finite list of threads is not practical. Since we cannot consider all threads when creating an ordering, there must be a maximum number of threads which can be taken into consideration without causing unacceptable delays. Let N_{Decide} be the number of threads to be considered at a time. This is essentially taking the infinite stream case and breaking it down into a series of finite streams.

We now have two options about what to do after we order the first N_{Decide} elements. First, we could execute all of them as scheduled, or execute the first $N_{Execute}$ of them, add the next $N_{Execute}$ that were not previously considered into the group to consider and repeat. Note that if $N_{Execute} = N_{Decide}$ then these two options are the same. Also, the first option requires less processing, while the second may provide better interaction in the cache. There is a downfall to this second option though. There is the potential for starvation. One possible fix is to take the *age* of a thread (i.e. how many times it has been in the scheduled portion of the list but not executed) into account when scheduling threads.

To decide the order in which to execute the $N_{Execute}$ threads we propose the simple metric of *Shared Hits* – *Shared Misses*, which may be extended in future research. These statistics can be either fed in initially to the system or collected over time. Note that unless the workload is very clearly defined these statistics will not be 100% accurate, but rather more of an average of the interaction of the cache footprints of the threads over time. We then do N_{Decide}^2 pair wise comparisons to pick the optimal order. In general, this exhaustive approach may be too time consuming, and it may be better to use a less precise method of picking the order of the threads. These are some of the questions we plan to explore in future work.

6 Conclusions

As we move into the multicore era, fine-grain multithreading will be among the primary tools used by software developers to enhance the parallel performance of their applications. However, more so than in prior, coarse-grain multithreading systems, intelligent locality-aware scheduling of these fine-grain threads is paramount for good performance.

In this paper, we have demonstrated the importance of preserving memory access locality across fine-grain threaded applications by experimentally evaluating two sets of synthetic workloads. One set involved a se-

ries of array-based workloads that showed a worst-case 5X performance difference between locality-optimal and locality-indifferent executions. The second set involved a series of tree-based traversal workloads that showed in excess of a 2X performance difference across best-case and worst-case executions.

We have also presented proposed architectural enhancements to allow the exposure of hardware-level thread footprint information to software-level systems to enable efficient and informed thread scheduling decisions. Using these proposed enhancements, we have also described a footprint-based thread scheduling algorithm to be evaluated in future work.

Footprint-based thread scheduling is a hitherto unexplored research domain that we have identified as holding great promise for future endeavor. Meanwhile, scheduling for fine-grain multithreading is an enterprising research area ripe for lasting contributions. Let there be much continued research that follows in these footsteps.

References

[1] T. E. Anderson, D. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. In *SIGMETRICS '89: Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 49–60, 1989.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, October 2001.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[4] M. de Kruijf and K. Sankaralingam. Mapreduce for the Cell B.E. Architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007-1625*, October 2007.

[5] K. Debattista, K. Vella, and J. Cordina. Wait-free cache-affinity thread scheduling. *Software, IEE Proceedings*, 150(2):137–146, April 2003.

[6] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *SIGMETRICS '88: Proceedings of the 1988 ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, pages 63–72, 1988.

[7] D. R. Engler, G. R. Andrews, and D. K. Lowenthal. Shared filaments: Efficient support for fine-grain parallelism on shared-memory multiprocessors. *Dept. of Computer Science. University of Arizona, TR 93-13*, April 1993.

[8] S. Harizopoulos and A. Ailamaki. Stageddb: Designing database servers for modern hardware. *IEEE Data Engineering Bulletin*, 28(2):11–16, June 2005.

[9] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, May 2007.

[10] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[11] J. Larus and M. Parkes. Using cohort scheduling to enhance server performance. *Technical Report MSR-TR-2001-39, Microsoft Research*, March 2001.

[12] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[13] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4, pages 131–143, February 1993.

[14] D. Thiebaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, 1987.

[15] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel Distributed Computing*, 24(2):139–151, 1995.

[16] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 230–243, 2001.