Thread Migration to Improve Synchronization Performance

Srinivas Sridharan, Brett Keck, Richard Murphy, Surendar Chandra, Peter Kogge
CSE Department, University of Notre Dame
384 Fitzpatrick Hall, Notre Dame, IN 46556
{ssridhar, bkeck, rcm, surendar, kogge}@nd.edu

Abstract

A number of prior research efforts have investigated thread scheduling mechanisms to enable better reuse of data in a processor's cache. We propose to exploit the locality of the critical section data by enforcing an affinity between locks and the processor that has cached the execution state of the critical section protected by that lock. We investigate the idea of migrating threads to the "lock hot" processor, enabling the threads to reuse the critical section data from the processor's cache and release the lock faster for other threads. We argue that this mechanism should improve the scalability of performance for highly multithreaded scientific applications. We test our hypothesis on a 4-way Itanium2 SMP running the 2.6.9 Linux kernel. We modified the Linux 2.6 kernel's O(1) scheduler using information from the Futex (Fast User-space muTEX) mechanism in order to implement our policy. Using synthetic micro-benchmarks, we show 10-90% performance improvement in cpu cycles, L2 miss ratios and bus requests for applications that operate on significant amounts of data inside the critical section protected by locks. We also evaluate our policy for the SPLASH2 application suite.

Keywords: *SMP, Multithreading, Synchronization, SMP Scheduler, Thread Migration, Profiling and tuning applications, Intel®Itanium2, Linux 2.6 Kernel, SPLASH2 suite*

1 Introduction

Scheduling threads on shared memory multiprocessors (especially SMPs) based on cache affinity has been previously investigated in [7], [17], [16] and [14]. These proposals attempt to efficiently reuse the thread's state that is already in a processor's cache by enforcing an affinity between the processor and the threads executing on them. In this paper, we apply this idea to locks and data in critical sections protected by these locks. The locality of the critical section data can be exploited by enforcing an affinity between locks and the processor that has cached the execution state of the critical section protected by that lock. We investigate the idea of migrating threads to the "lock hot" processor, enabling the threads to reuse the critical section data from the processor's cache and release the lock faster for other threads.

The main advantage of this approach is that since we are enabling threads to be closer to the execution state accessed within a critical section. This technique minimizes costly cache misses and improves the cache locality awareness of multithreaded scientific applications. Further, moving threads that do not require the lock away from the processor enables the thread holding the lock to complete its execution faster and release the lock quicker for other contending threads. Finally reduction in cache misses also translates to reduction in number of requests on the communication network (shared bus in the case of small scale SMP) and indirectly helps improve the scalability of the system by adding more processors.

We argue that our policy should improve the performance for highly multithreaded scientific applications. To test our hypothesis we used a 4-way Itanium2 SMP running the 2.6.9 Linux kernel. We have extensively leveraged the Linux 2.6 kernel's O(1) scheduler and Futex (Fast User-space muTEX) mechanism in order to implement our ideas. The O(1) scheduler, which takes constant time irrespective of the number of threads and the Futex mechanism, which implements the OS support for user-level locks, provide significant performance/scalability improvements to multithreaded applications over the earlier 2.4 kernel. Our experimental suite consists of seven hand-coded microbenchmarks and nine kernels/applications from the SPLASH2 suite. The microbenchmarks show a 10-90% performance improvement in cpu cycles, L2 miss ratios and bus requests for applications that operate on significant amounts of data inside the critical section protected by locks. The SPLASH2 benchmarks also show significant performance improvements for L2 miss ratios and bus requests for most applications in the suite.

The remainder of the paper is organized as follows: Section 2 presents details on scheduling and synchronization

techniques under and presents some details on the Linux Kernel scheduler. Section 3 presents the experimental setup and Section 4 presents the results of our evaluation. Section 5 summarizes related work, Section 6 presents our conclusions and suggests some future work.

2 Scheduling and Synchronization techniques

In this section, we first present the conceptual overview of our new scheduling policy. Then we provide a brief explanation of the Linux 2.6.9 kernel scheduler and the futex mechanism. Finally, we present the modifications we made to the linux kernel to implement our policy.

2.1 Conceptual Overview

Our OS level synchronization technique relies heavily on the kernel thread scheduler and requires the scheduler to be aware of user-level locks. In other words, the kernel scheduler must be able to identify the thread that currently holds a contended lock. Using this information, the kernel can pin the thread and its execution state to a single cache for reasonable amounts of time. The locality of the critical section data can be exploited by moving threads that require the same lock to the processor that is executing the thread that currently holds the lock. In addition to this, we allow the scheduler to move threads that do not require the lock to other processors in the system, reducing the workload on the processor whose cache holds the lock.

There are number of advantages in doing this. For example, when a lock is released and the thread that blocked on the lock is awakened, it directly uses the data in its local cache rather than doing remote bus requests to fetch the data. This is quite important given the fact that current micro-processors are much faster than the memory subsystem and the system bus, and hence have to use the data in their caches very efficiently. Additionally this translates to reduction in the number of requests in the bus and hence improvement in the scalability of SMP systems. Further the thread is able to perform the computation in the critical section faster and release the lock quicker since the data is already present in its local cache. All these advantages should speed up applications that have heavy synchronization overhead.

2.2 Linux 2.6.Kernel Scheduler and Synchronization support

The Linux kernel does not differentiate between processes and threads as most operating systems do. Any reference to tasks or threads refer to the same entity since we are primarily interested in multithreaded applications.

2.2.1 Scheduling

The Linux 2.6 kernel scheduler uses a O(1) algorithm i.e. the scheduler is guaranteed to schedule tasks within a certain constant amount of time regardless of the number of tasks currently on the system. This is largely made possible by having separate run-queues for each physical processor in the system. The distributed run queues also enables the scheduler to provide better scheduling support for SMPs. First, the load on each processor is estimated by multiplying the number of active tasks on the run queue by a scaling factor. This metric enables the scheduler to handle load imbalance in the system. Further, this is also used to enforce better "SMP affinity" i.e. tasks stay on the same run queue (or processor) for longer so that they utilize the caches better.

These techniques provide significant improvement over the previous 2.4 kernel where threads *randomly* migrated across processors resulting in poor performance. In addition to these improvements, we propose to migrate tasks when they (un)block for synchronization events since they incur heavy serialization overheads.

The Linux 2.6 kernel scheduler uses a variety of techniques to schedule/migrate threads depending on the system state. One way that threads are scheduled is by the Migration thread which is a per CPU high priority kernel thread. When the load is unbalanced, the Migration Thread will migrate threads from a processor that is carrying a heavy load to a processor or processors that currently has light load. The migration thread is activated based on a timer interrupt to perform active load balancing or when requested by other parts of the kernel. Another way that scheduling is done is on a thread by thread basis. When each thread is unblocked or is being scheduled to run, the scheduler will check to see if it can run on its currently assigned processor, or if it needs to be migrated to another processor to keep the load balanced across all processors.

2.2.2 Synchronization

Since Linux 2.5.x kernel series, user level synchronization is supported in the OS by using the kernel Futex (Fast Userspace muTEX) mechanism. Futexes are light-weight, and can be used as building blocks for implementing fast user-level locks and semaphores in system libraries. For example, the Linux 2.6 POSIX thread library also called as NPTL (Native Posix Thread Library) uses futexes to implement pthread_mutex calls.

Operations on futexes start in the user-space but enter the kernel when necessary using the the `sys_futex` system call. The Linux man pages defines `sys_futex()` as: "...`sys_futex()` system call provides a method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address".

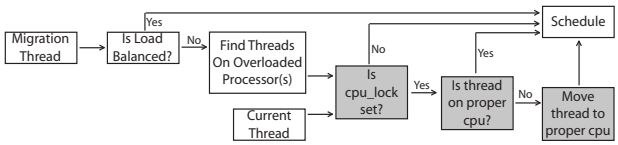

Figure 1. Migration flow within the Scheduler

In short the kernel futex mechanism works as follows:

- When a futex is free, a user-level thread can acquire the lock, but it does not need to enter the kernel to do so. Similarly, the thread need not enter the kernel for releasing an un-contended lock.
- When a futex is not free (lock is contended), a thread wishing to acquire the lock will enter the kernel. It is then queued along with all other previously blocked threads using the `sys_futex(..., FUTEX_WAIT, ...)` system call.
- When a thread releases a lock that is contended, it will enter the kernel and wake up one or more blocked threads on the corresponding futex using the `sys_futex(..., FUTEX_WAKE, ...)` system call.

For threads sharing the same address space, a futex is identified using the virtual address of the lock in the user-space. This address is used to map into a kernel data structure that implements a hashed bucket of lock addresses. The mechanism also provides support for inter-process communication, but the details of its working are beyond the scope of this paper.

2.3 Linux 2.6.9 Kernel Modifications

2.3.1 FAST: Futex Aware Scheduling Technique

The goal behind our modifications is to create a link between the kernel futex mechanism and the scheduler, in other words, to make the scheduler "futex aware". This additional knowledge enables the scheduler to make intelligent decisions for threads that are contending for locks. To do this, we modify three parts of the kernel: the thread information table, the futex implementation, and the scheduler.

Thread Information Table is unique to each thread. This table has an entry called `cpu` to represent the physical CPU on which the thread is currently on. We added a new entry to the table called `cpu_lock` to represent the physical CPU (lock hot processor) the thread will need to run on when it acquires a lock. Upon the creation of a new thread, this field is initialized to value larger than the number of processors in the system called `CPU_LOCK_INIT`. In our case, for a four-way SMP any value larger than four should work.

Futex Mechanism We added our modifications to the futex mechanism in the *futex_wake()* function which in turn gets invoked by the `sys_futex(..., FUTEX_WAKE, ...)` system call. When a thread is releasing a contended futex (releasing thread), it wakes up one of the threads already blocked (acquiring thread) on the futex. Here, we set the `cpu_lock` for the acquiring thread to that of the `cpu` of the releasing thread. As the releasing thread is no longer waiting for the lock, we set its `cpu_lock` value to `CPU_LOCK_INIT`.

Scheduler The scheduler checks the `cpu_lock` when migrating a thread or when trying to activate a blocked thread. If the thread is being migrated due to reasons not related to its synchronization activities (`cpu_lock=CPU_LOCK_INIT`), then the scheduler performs its operations as usual. In other cases, the scheduler's decisions is modified based on the following heuristics:

- If a thread is already running on the processor identified by the `cpu_lock` value, the thread is not migrated away from that processor.
- If a thread has a `cpu_lock` value defined (other than `CPU_LOCK_INIT`), migrate this thread to the processor identified by `cpu_lock`.
- In any case, if there is a problem with load balancing, the default scheduler thread migration mechanisms have higher priority over our policy.

Figure 1 shows a brief flow of control within the linux kernel scheduler. The shaded boxes represent additional decisions introduced by our scheduling policy. The kernel modifications we propose are non-intrusive to the normal operation of the scheduler as long as threads do not block for synchronization events such as contended locks. Additionally since only one thread is awakened at a time, the "lock hot" processor is not running heavy loads. Finally we also do not interfere with *which* of the blocked threads is awakened but we only change the decision of *where* it is awakened. The scheduler native support for load balancing ensures that the processors do not become unbalanced over time. Further, this also ensures that multiple locks do not get assigned to the same processor.

In terms of source code we have added 20-30 lines totally to both the scheduler and the futex mechanism. However since scheduler code is executed at high frequency the

Parameter	Details
Server name	HP Integrity rx4640-8 Server
SMP Type	4-way Intel®Itanium®2
Processor Speed	1.5 GHz
L1I cache size (line size)	16Kb (64 Bytes)
L1D cache size (line size)	16Kb (64 Bytes)
L2 cache size (line size)	256KB (128 Bytes)
L3 cache size (line size)	4MB (128 Bytes)
Memory size	8 GB
Bus bandwidth	12.8 GB/s
OS kernel (Distribution)	Linux 2.6.9 kernel (Redhat Linux Enterprise)

Table 1. Baseline system parameters

number of lines of source code may not represent the actual overheads. Further, preliminary analysis shows that we have not modified the O(1) scheduling guarantees of the scheduler. Theses issues are being further analyzed as part of the future work.

3 Experimental Methodology

This section gives details on the experimental methodology used in this study. The baseline system used for all the experiments is presented in Table 1. The rest of this section is organized as follows: first, we explain the performance monitoring tools and metrics we used on the Linux IA64 environment. Next, we provide details on the microbenchmarks and applications that were used to evaluate the effect of our modifications.

3.1 Performance Monitoring tools and Metrics

The Pfmon utility [1] is a performance monitoring tool used to collect counts or samples from unmodified binaries running on IA64 processors (Itanium, Itanium2). It can also be used to profile an entire system. It uses the IA-64 hardware performance counters and the Linux kernel perfmon interface to monitor real applications. It makes full use of the libpfm library to help in programming the IA-64 Performance Monitoring Unit (PMU) [2].

For our experiments, we monitored the following hardware events: CPU cycles (CPU_CYCLES), number of bus requests (BUS_ALL_SELF), L2 cache misses (L2_MISSES), L2 cache references (L2_REFERENCES), L3 cache misses (L3_MISSES), L3 cache references (L3_REFERENCES). The L2 and L3 miss ratios were obtained from the ratio of the respective misses and reference numbers. Further each thread was monitored individually and the metrics we aggregated over the values of all the threads. Finally, we monitored the above events for both user and kernel space.

3.2 Microbenchmarks and Benchmark Suites

3.2.1 Microbenchmarks

We developed seven microbenchmarks that specifically test synchronization mechanisms and implemented them in C language using Pthreads (POSIX Threads Library). The seven microbenchmarks differ on a wide range of characteristics including the number of locks, the access patterns of shared data structures with the critical section, the complexity of the critical section code and finally optimizations that try to limit false sharing.

The first four microbenchmarks all use one lock to protect the critical section and the last three involve multiple locks. All the threads execute the critical section in a loop, with iterations ranging from 200,000 to 16 million. Further, all the microbenchmarks are padded appropriately to take care of false sharing. All microbenchmarks perform significant amounts of random work outside the critical section code. This is to ensure fairness in acquiring the lock and performing critical section operations. In other words threads dont iteratively just acquire/release locks but do perform some work in between successive locking operations.

All the microbenchmarks were designed using a methodology similar to [13] and [11]. Four of the microbenchmarks (Single counter, Multiple counter, Doubly linked list, Producer Consumer) were previously mentioned in [8] and [13]. We explain each of these microbenchmarks briefly.

Single Counter(single_ctr) consists of a counter (fits in a cache line) protected by a single lock and all threads increment a single counter in a loop.

Multiple Counter(multiple_ctr) consists of an array of counters protected by a single lock and each thread increments a different counter (fits in a cache line) in the array.

Doubly Linked list(doubly_list) consists of a doubly linked list where threads dequeue elements from the head and enqueues them on to the tail of the list. The enqueue and dequeue operations are performed independent of each other with separate lock acquire and release operations. The doubly linked list is protected by a single lock.

Producer Consumer(prod_cons) consists of a shared FIFO (bounded) array protected by a single lock that is initially empty. Half the threads produce items into the FIFO that are consumed by the other half threads. Producers have to wait for free elements in the FIFO

Benchmark	Problem size
Cholesky	d750.o
FFT	2^{24} points
LU-Cont (Contiguous)	2,048x2,048 matrices
LU-Noncont (Noncontiguous)	2,048x2,048 matrices
Radix	2^{24} integers, radix 1024
Barnes	92,000 bodies
FMM	32,000 particles
Water-nsq (Nsquared)	9,261 molecules
Water-spa (Spatial)	9,261 molecules

Table 2. SPLASH2 problem sizes

whereas consumers have to wait for data to consume before iterating the critical section code.

Affinity Counter(affinity) consists of two locks that protect two counters. During the first phase of each iteration, each of the locks protects one of the counters and during the second phase each of the locks protect the other counter. A barrier is required between the two phase to ensure atomicity.

Multiple Counters Multiple Locks(mlt_ctr_mlt_lock) is similar to the multiple counter microbenchmark, but there are multiple locks protecting different segments of the counter array. The threads dynamically choose the lock to acquire and hence the counter to update depending on its thread id.

Multiple FIFO(multiple_fifo) is similar to producer consumer microbenchmark, but there are multiple locks each protecting a separate FIFO. Each producer/consumer pair is dynamically assigned to a FIFO depending on its thread id. Each thread acquires/releases only the lock corresponding to the queue it is assigned irrespective of whether it is producing or consuming data.

3.2.2 SPLASH2 Benchmark Suite

SPLASH2 (Stanford Parallel Applications for Shared Memory) [18] consists of five kernels and eight applications to test various characteristics of shared memory machines. SPLASH2 codes utilize the Argonne National Laboratories (ANL) parallel macros (PARMACS) for parallel constructs. We used the Pthreads implementation of PARMACS by [3]. We used nine of the SPLASH2 kernels/applications with the problem sizes listed in Table 2. Our problem sizes is larger than the default problem sizes listed as part of SPLASH2 documentation. This was necessary since the base problem sizes supported only 64 processors/threads.

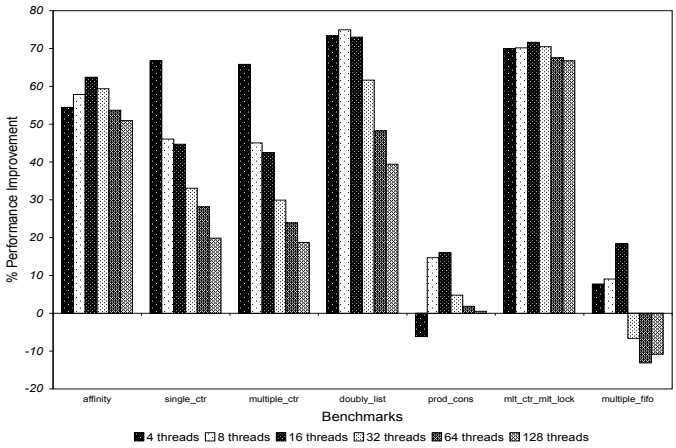

Figure 2. Microbenchmarks: % Improvement in number of CPU cycles for modified kernel over unmodified kernel

4 Results

4.1 Microbenchmarks results

Each of the application/thread combination was run five times and the results presented in these graphs represent the data averaged over all the runs. For each benchmark, we measure the percent improvement for 4, 8, 16, 32, 64, and 128 threads. These values were obtained using pfmon with metrics such as CPU_CYCLES, L2_MISSES, L2_REFERENCES, L3_MISSES, L3_REFERENCES, BUS_ALL_SELF and were calculated using the formula: $(((Base\ kernel\ value - Modified\ kernel\ value) * 100)/Base\ kernel\ value)$.

Figure 2 shows the percent improvement in cpu cycles for the seven microbenchmarks in the modified kernel over the unmodified kernel. For example, in the case of 128 threads, the *affinity* shows an improvement of 50.9% implying that the modified kernel ran this benchmark in half the number of cpu cycles as the unmodified kernel. The best improvements were obtained for the 4, 8, and 16-thread cases, with peak improvements as high as 99%. Overall, an improvement of 18-75% is observed for five of our seven microbenchmarks (affinity, single_ctr, multiple_ctr, doubly_list, mlt_ctr_mlt_lock) for the range of threads under consideration. The 4-thread case for prod_cons, as well as the 32, 64, and 128-thread cases for multiple_fifo, do not show an improvement. This is primarily due to the scheduling overhead being higher than the synchronization overhead for these cases.

Table 3. Microbenchmarks: % Improvement in L2 and L3 miss ratios for modified kernel over unmodified kernel

Micro Benchmark	Threads					
	4	8	16	32	64	128
Affinity	4.4	**54.4**	18.8	29.4	30.5	27.9
Counter	94.3	98.6	**98.9**	98.6	97.7	83.5
Single	**34.8**	11.4	17.5	13.8	10.9	10.5
Counter	-0.6	0.1	**0.4**	-4.3	-5.1	-1.2
Multiple	**36.4**	14.9	16.9	15.2	11.7	9.1
Counter	-1.3	0.2	**1.2**	-3.9	-4.6	-3.0
Doubly	**42.0**	29.8	31.7	21.2	22.7	15.2
Linked list	-2.3	0.4	**7.4**	2.0	-0.9	-4.4
Producer	**40.2**	20.5	10.8	10.2	9.5	8.3
Consumer	-0.9	0.5	**1.9**	-0.6	-4.9	-1.5
Mlt. Ctr	**59.4**	30.5	34.0	36.8	35.4	36.7
Mlt. Lock	-0.1	4.9	**16.4**	10.0	2.4	0.1
Multiple	**34.0**	18.8	9.0	7.9	6.0	6.0
FIFO	-0.8	0.5	**2.9**	-2.8	-4.3	-2.0

The percent improvement in the L2 and L3 miss ratios is listed in Table 3. For each benchmark, the first row represents the L2 miss ratios and the second row represents the L3 miss ratios. The boldface numbers represent the best case percentage improvements for the L2 and L3 miss ratios. All microbenchmarks show an improvement of 6-94% for the L2 miss ratio. This improvement is a key artifact of our modification, translating to a reduction in the number of bus requests, cpu cycles, and L3 cache references.

However, it should be noted that the L3 miss ratios do not show these same improvements, with a small negative improvement in many cases. The lower L2 miss ratio generates a lesser number of L3 references in the modified kernel compared to the unmodified kernel. Cold-start misses will still occur, and the combination of these L3 cache misses and the lower number of L3 references result in a higher L3 miss ratio for the modified kernel in many cases.

Figure 3 shows the percentage improvement in bus requests for the microbenchmarks. As in the cpu cycles case, the 4, 8, and 16-thread cases performed best, with improvements ranging from 10-99%. In general, significant performance improvements were observed until 16 threads after which the performance improvements started decreasing due to system load and other overheads. Again, the 32, 64, and 128-thread cases of multiple_fifo did not show improvement, as the overhead from scheduling this program will outweigh the benefit created from the synchronization improvement.

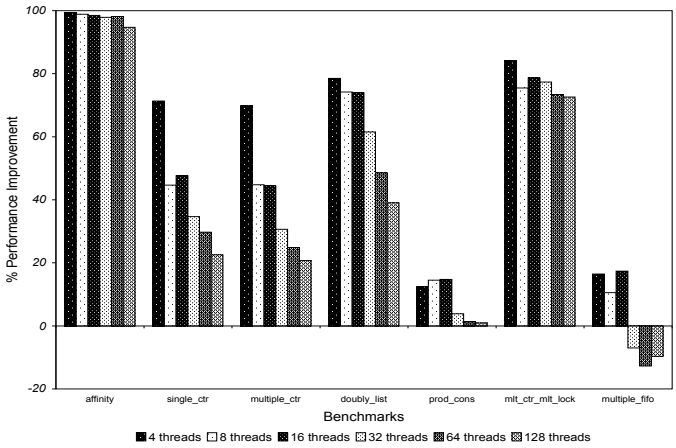

Figure 3. Microbenchmarks: % Improvement in number of requests on the bus

The *affinity* microbenchmark performs best across all metrics, as it is specifically designed to create explicit affinity between the locks and the data accessed within the critical section. In the unmodified kernel, since locks and critical section data are not pinned down to a particular cache, the cache lines get transferred across processors more frequently.

4.2 SPLASH2 Results

Table 4 shows the improvements in the L2 miss ratios for SPLASH2 codes. We see improvements ranging from a negative 5% to a positive improvement of 33%, with most cases showing positive improvements. As in the case of the microbencharks, the boldface numbers represent the best case percentage improvements for L2 miss ratios. Figure 4 shows the percentage improvements for number of bus requests. In general, the results show a fairly consistent trend of providing better results for a higher number of threads with increase in available parallelism. Figure 5 shows the performance improvements for cpu cycles. The performance improvements range from a negative 39% to a positive improvement of 10%, with most applications having little or no performance improvements in terms of cpu cycles.

We are currently investigating why some SPLASH2 applications do not show improvements as well as others, and why many do not perform in a consistent basis. Specifically, we are trying to determine why the improvements in L2/L3 miss ratios and the number bus requests have not translated to corresponding improvements in cpu cycles.

Table 4. SPLASH2: % improvement in L2 miss ratios for modified kernel over unmodified kernel

Benchmark	Threads					
	4	8	16	32	64	128
Barnes	2.1	4.7	6.6	8.5	16.4	**21.7**
Cholesky	0.8	0.5	0.7	**2.4**	0.8	1.0
FFT	0.0	0.5	0.7	0.6	**3.9**	3.8
FMM	-0.7	0.4	0.3	1.8	-0.7	**2.7**
LU-Cont	-5.6	0.2	1.3	3.5	**4.4**	3.3
LU-Noncont	**33.5**	17.3	9.8	9.6	8.1	5.2
Radix	1.1	6.3	14.5	21.5	20.8	**21.6**
Water-Nsq	0.4	0.0	-0.2	-0.4	**0.5**	-0.7
Water-Spa	1.3	-1.2	-1.1	**5.5**	5.3	3.4

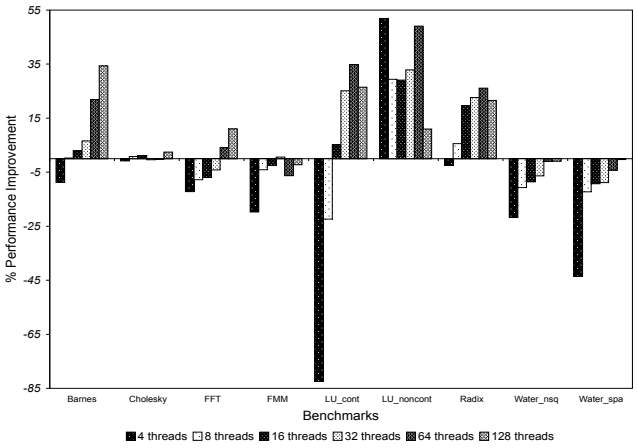

Figure 4. Splash2: % Improvement in number of bus requests for modified kernel over unmodified kernel

If benchmarks do not spend a high percentage of time performing synchronization, then the scheduling overhead introduced by our modification is will not offset by the application's synchronization overhead. We also conjecture that SPLASH2 is tuned for 32 bytes cache block size instead of the processors 128 byte block size. Further on the Itanium processors pointers are 64-bit values. So if an application mixes pointers and scalar data types inconsistently without cache line sizes in mind then the application takes a significant hit. Such issues may be even more important with our modifications. We intend to investigate these issues more thoroughly as part of future work.

5 Related Work

A variety of hardware and software techniques have been proposed for supporting efficient synchronization. These techniques allow efficient synchronization to be implemented in user-level thread libraries [12], parallelizing compilers, OS runtime [4] [5], atomic instructions such as Test&set[9], memory consistency models [6], hardware synchronization protocols [10], and cache coherency protocols [15] etc. We direct interested readers to the above references for more information on the implementations. Evaluation of synchronization mechanisms have also been studied both by simulation [10] and on real systems [11], [12]. We use the methodology presented in [11] and some of the microbenchmarks used in this paper have previously been used by [13] and [8].

The idea of cache affinity scheduling has been previously explored in [7], [17], [16] and [14]. These proposals have been evaluated using analytical models, simulations and on real systems using both synthetic and real workloads. Simi-

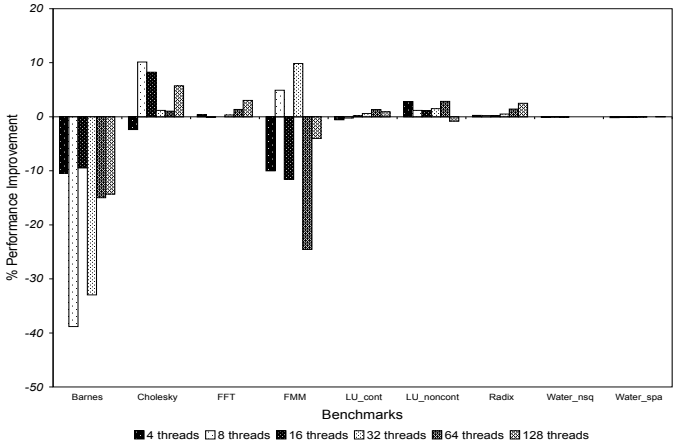

Figure 5. SPLASH2: % Improvement in number of CPU cycles for modified kernel over unmodified kernel

lar to the previous papers we have tried to leverage the execution state that is already present in caches to speedup our applications. However unlike the previous approaches we primarily use synchronization events to guide our heuristic.

6 Conclusions and Future Work

In this paper, we exploit the locality of the critical section data by enforcing an affinity between locks and the processor. Using microbenchmarks, we show the potential for large performance gains (up to 90%). We also show that when an application is properly written with cache line size in mind, we can further decrease the L2 miss ratio, and in turn lower the L3 miss ratio, the number of bus requests, and the cpu cycles for an application. Our scheduler helps improve the performance for highly multithreaded scientific applications.

We are currently working on an online mechanisms that can identify circumstances under which our approach works best and then dynamically switches the scheduler to use our policy only for those cases. We are continuing our efforts to optimize the scheduler by performing finer-grain traces of the scheduler in order to fine-tune it and limit the overhead introduced by our modifications. We also plan to test our scheduling techniques on multi-core SMP systems, as we hypothesize that our modifications could allow for even larger improvements on such a system. Finally, we plan to extend our testing to a wide variety of applications and testing suites, so we can gain a better understanding of the opportunities and limitations of our approach.

Acknowledgments

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under its Contract No. NBCH3039003. The Itanium2 servers were provided by a grant from HP with additional support from NSF (CNS0447671, IIS0515674).

References

[1] Perfmon project. http://www.hpl.hp.com/research/linux/perfmon/pfmon.php4.

[2] Intel®Itanium®2 Processor Reference Manual for Software Development and Optimization. http://www.intel.com/design/itanium2/manuals/251110.htm, May 2003.

[3] E. Artiaga, N. Navarro, X. Martorell, and Y. Becerra. PARMACS Macros for Shared-Memory Multiprocessor Environments. Technical Report UPC-DAC-1997-07, Department of Computer Architecture, UPC, Jan. 1997.

[4] U. Drepper. Futexes are tricky. http://people.redhat.com/drepper/futex.pdf.

[5] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, 2002.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, New York, NY, USA, 1990. ACM Press.

[7] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 120–132, New York, NY, USA, 1991. ACM Press.

[8] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural support for lock-free data structure. In *Proceedings of the 20th ISCA*, pages 289–300. ACM Press, 1993.

[9] I. B. M. I. Inc. IBM System/360 Principles of Operation. USA, May 1970.

[10] A. Kagi, D. Burger, and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, pages 170–180, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[11] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh. Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance. *SIGMETRICS Perform. Eval. Rev.*, 27(1):23–34, 1999.

[12] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions Computing Systems*, 9(1):21–65, 1991.

[13] R. Rajwar and J. Goodman. Transactional Lock-free Execution of Lock-based Programs. In *Proceedings of the 10th ASPLOS*, pages 5–17. ACM Press, 2002.

[14] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):131–143, 1993.

[15] P. Stenstrom, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 109–118, New York, NY, USA, 1993. ACM Press.

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

[17] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 26–40, New York, NY, USA, 1991. ACM Press.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.