Analyzing Lock Contention in Multithreaded Applications

Nathan R. Tallent
Rice University
tallent@rice.edu

John M. Mellor-Crummey
Rice University
johnmc@rice.edu

Allan Porterfield
Renaissance Computing Institute
akp@renci.org

Abstract

Many programs exploit shared-memory parallelism using multithreading. Threaded codes typically use locks to coordinate access to shared data. In many cases, contention for locks reduces parallel efficiency and hurts scalability. Being able to quantify and attribute lock contention is important for understanding where a multithreaded program needs improvement.

This paper proposes and evaluates three strategies for gaining insight into performance losses due to lock contention. First, we consider using a straightforward strategy based on call stack profiling to attribute idle time and show that it fails to yield insight into lock contention. Second, we consider an approach that builds on a strategy previously used for analyzing idleness in work-stealing computations; we show that this strategy does not yield insight into lock contention. Finally, we propose a new technique for measurement and analysis of lock contention that uses data associated with locks to blame lock holders for the idleness of spinning threads. Our approach incurs $< 5\%$ overhead on a quantum chemistry application that makes extensive use of locking (65M distinct locks, a maximum of 340K live locks, and an average of 30K lock acquisitions per second per thread) and attributes lock contention to its full static and dynamic calling contexts. Our strategy, implemented in HPCTOOLKIT, is fully distributed and should scale well to systems with large core counts.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Measurement techniques, Performance attributes; D.1.3 [*Programming techniques*]: Concurrent Programming—Parallel programming

General Terms Performance, Measurement, Algorithms.

Keywords Performance Analysis, Lock Contention, Multithreading, HPCTOOLKIT.

1. Introduction

Many programs exploit shared-memory parallelism using multithreading based on thread libraries such as POSIX Threads (Pthreads) [6]. Despite a recent surge of interest in transactional memory [18], locks remain the principal mechanism used to guard the integrity of shared data structures in multithreaded programs. In fact, some of the fastest software implementations of transactional memory use locks under the hood [11].

PPoPP'10, January 9–14, 2010, Bangalore, India.

Contention for locks has long been recognized as a key impediment to performance for shared-memory parallel programs. Early simulation studies of large-scale shared-memory parallel systems showed that hot spots, such as those caused by spin-waiting for locks on machines without coherent caches, could dramatically degrade performance by clogging multistage interconnection networks [21]. Later work explored alternative implementations for locks that reduce interconnection network traffic associated with spin-waiting, *e.g.*, [1, 19]. Today, the potential for performance losses in parallel systems due to synchronization traffic resulting from spin-waiting is well understood and in most cases it can be largely avoided by using appropriate algorithms.

However, a fundamental performance problem caused by using locks in parallel programs and run-time systems remains: contention for locks causes serialization. As a result, idling while waiting for a lock reduces parallelism and parallel efficiency. For this reason, pinpointing and ameliorating sources of lock contention in parallel applications is of significant interest. As the number of cores per processor increases, the scale of multithreading will grow. Diagnosing performance bottlenecks in multithreaded applications will be of increasing interest as multithreaded applications become ubiquitous. A tool that helps pinpoint sources of lock contention and quantifies their performance impact can provide invaluable guidance for tuning multithreaded applications.

This paper proposes and evaluates three strategies that a performance tool can use to gain insight into performance losses due to lock contention. The approaches we consider move from blaming lock contention on victims, then to suspects, and finally to perpetrators. This shift in perspective can be subtle — the first two strategies are actually modest extensions to state-of-the-art measurement techniques — but it is critical. Section 2 explores the utility of attributing the idleness of spin-waiting for locks directly to the calling contexts in which spin-waiting occurs (victims). Section 3 considers spreading the blame for idleness due to lock spin-waiting among threads holding locks (suspects). Section 4 describes a new strategy for directly blaming a lock holder for the idleness of threads spinning on a lock that it holds (perpetrators).

We evaluate our new strategy of directly attributing blame for lock contention in Section 5. We use three codes: MADNESS [16] — a quantum chemistry application that makes extensive use of locking; UTS [20] — an unbalanced tree search benchmark; and SSCA #2 [4] — a graph analysis benchmark that is a member of the Synthetic Scalable Compact Application Benchmark suite [9]. For complex applications like these, locks may be acquired frequently — an execution of MADNESS uses 65M distinct locks, a maximum of 340K live locks, and an average of 30K lock acquisitions per second per thread — and the sources of lock contention can be context sensitive. Moreover, a performance tool must not itself significantly affect an execution. This is difficult to ensure. Adding overhead to critical sections can make the tool itself a new source of contention, while adding overhead outside of critical sections can *reduce* contention. Consequently, any tool for understanding

lock contention must operate with very low overhead, obtain calling context, and produce insightful metrics. The significance of our result is that we achieve *all* these goals.

Finally, Section 6 relates our strategies to prior work; and Section 7 offers some conclusions from our studies.

2. Attributing idleness to its calling context

2.1 A straightforward strategy

The first strategy we consider for understanding the impact of lock contention in multithreaded programs is straightforward and is based on two key ideas.

The first idea is to quantify lock contention by measuring lock idleness, *i.e.*, the idle time a thread spends waiting for a lock. Thus, we distinguish between the useful work that a thread performs and its idleness. If a thread repeatedly idles waiting for a lock, then its idleness metric will consume a significant percentage of the thread's total effort (effort = work + idleness).

The second idea is to use *call path profiling* [14] to attribute these metrics to the calling context in which they are incurred. Call path profiling is especially useful for modular programs, where it is important to attribute costs incurred by procedures to the different contexts in which the procedures are called. We use HPCTOOLKIT's sampling-based call path profiler [24] that attributes metrics to the full static and dynamic contexts in which they are incurred. Sampling-based call path profilers use a recurring event trigger to raise signals within the program being profiled. When an event trigger occurs, it raises a signal, and a signal handler obtains a call path by unwinding the call stack. HPCTOOLKIT's profiler incurs minimal overhead for reasonable sampling rates (typically 2–3% for 200-1000 samples/second) and is capable of measuring and attributing performance metrics to fully optimized code.

To combine these two ideas, when attributing a sample to its calling context, it is necessary to know whether the sample represents work or idleness. Consider the case of 'right-sized' parallelism, where each thread is associated with a unique hardware context. In this case, threads would typically use spin locks, *i.e.*, locks that busy-wait rather than yield to the operating system (OS). Since each thread has a sample source, samples are delivered to a thread both while it is working and while it is spinning for a lock. To determine whether to charge a sample to a work or idleness metric, we intercept a monitored application's calls to lock routines to set a thread-local flag immediately before and after the thread begins waiting for a lock. In contrast to samples, which arrive asynchronously and whose frequency can be controlled independently of the application, this flag is set *synchronously* on *every* lock attempt. Keeping instrumentation overhead low is important; the cost of having locking routines maintain a flag is not a problem.

2.2 Blocking (sleep-waiting)

In contrast to spin locks, Pthreads mutex locks and condition variables sleep-wait. When a thread is sleeping, no user-level resources are used, effectively muting any sampling triggers based on those resources.[1] An obvious solution to the problem at hand is to directly measure lock (or condition variable) wait time. However, this requires gathering time stamps both before and after a wait and, if the idleness is non-zero, attributing it to the calling context. Thus, it is potentially necessary to perform an unwind for *every* lock release, which would cause significant overhead for programs that have a high volume of lock acquisitions and releases. Applying this strategy to measure locking in MADNESS [16] (see Section 5.1), which performs 30K lock acquisitions per second per thread, yielded a monitoring overhead of 260%. To reduce this overhead, we can 'sample' the lock acquisitions themselves. That is, on every n^{th} lock acquisition, measure the thread's idleness I and attribute $n \times I$ units of idleness to the calling context. In effect, this scheme amortizes the cost of heavyweight instrumentation across n lock acquisitions.

[1] It is possible to use a sampling trigger based upon real time rather than user time, but on standard OS's, this does not work well with threads. For example, on Linux, `ITIMER_REAL` does not provide a thread-specific sample source and therefore delivers signals to a random thread within a process.

2.3 Evaluation

For the Pthreads library, we implemented this strategy by *overriding* routines that could potentially cause a thread to idle: `pthread_{spin,mutex}_lock` and `pthread_cond_wait`. To override a routine in a dynamically linked application, we use library preloading.[2] That is, at program launch time, HPCTOOLKIT injects a dynamically linked profiling library into an unmodified program's address space. For statically linked programs, compilation remains unchanged, but we require users to adjust their link step to invoke a script that adds HPCTOOLKIT's profiling library to a statically linked executable.[3] When a monitored application calls one of the overridden routines, control is transferred to the monitored version of the routine, or the *override*. The override then sets a thread-local idleness flag — pessimistically assuming the thread will idle — and immediately calls the actual Pthreads routine. When the thread enters the lock or condition variable critical section, the Pthreads routine returns to the override, which immediately clears the idleness flag and returns to the monitored application.

This strategy computes a thread's idleness with accuracy and with low overhead. On average, a thread receives samples while its idleness flag is set in proportion to the time it is actually idle. If a thread attempts to acquire a lock many times but without contention, that thread will spend relatively little time with its idle flag set and its idleness metric will be proportionately small. In contrast, if a thread spends a large percentage of time idle, whether due to few or many lock acquisitions, its idleness metric will proportionately reflect this fact. Consequently, our conservative assumption yields a simple implementation without sacrificing accuracy. Another important benefit of this scheme is that all data is thread-local which means that it naturally scales to a large number of threads.

One limitation of our implementation is that it does not handle over-subscription — *i.e.*, when there are more threads than available hardware contexts — if a thread sleep-waits.

The more serious limitation of this approach is that it fails to yield the insight into lock contention that we desire. While this idleness metric reflects contention in the sense that higher contention results in higher idleness, it pinpoints the symptom rather than the cause; the victim rather than the perpetrator. In other words, this idleness metric takes a "first person" view of lock contention and records its effect rather than its provenance by blaming a waiting thread for its own waiting. To pinpoint the cause of idleness, idle threads must have some "third person" knowledge about which threads are responsible for their idleness. We next describe an idleness metric that attempts to account for this problem.

3. Blaming idleness on lock-holders

3.1 Extending a prior strategy

In prior work, we recognized the problem of attributing idleness as a symptom rather than as a problem source [23]. In that work, we described an idleness metric that blamed idleness in work-stealing programs to regions of code with too little parallelism. In Cilk [12], such parallelism is expressed with asynchronous calls.

[2] On Linux, see the loader's special environment variable `LD_PRELOAD`.

[3] On Linux, see the linker's special `--wrap` option.

We implemented our ideas by modifying the Cilk run time to 1) track when an individual thread was working or idle; and 2) maintain a node-wide counter representing the total number of working (W) and idle (I) threads. Like the strategy of Section 2, if a sample event occurs in a thread that is actively working, the thread attributes that sample to a work metric associated with the sample context. However, there are two key differences. First, the working thread also attributes a fractional sample I/W to an idleness metric associated with the sample context to blame itself for the current idleness in the execution. Second, if a sample occurs in an idle thread, it is simply ignored. This strategy equally spreads the blame for not keeping threads busy at that moment to the active contexts of working threads.

This strategy can be adapted to Pthreads. As in Section 2, we override Pthreads routines that potentially cause a thread to idle (`pthread_{spin,mutex}_lock` and `pthread_cond_wait`). We add a node-wide counter to maintain the number of working threads, W. During an override, immediately before calling an actual Pthreads library primitive that might wait, we atomically decrement W; we then increment W when the primitive returns. At any point in time, I can be computed implicitly as $T - W$, where T is the number of threads. Then we process samples as described above.

One natural benefit of this strategy is that there is no need to distinguish between spin-waiting and sleep-waiting. In the first strategy it was necessary to handle sleep-waiting specially (using timers) because sleeping threads do not receive samples. However, in this scheme, any samples received by an idle thread are already ignored.

Although our prior work suggested that this strategy could be effectively applied to Pthreads, we found that it did not yield actionable insight into lock contention within complex applications like MADNESS. There is a simple explanation for why evenly apportioning blame is not very useful for a threaded application using locks. For a work-stealing scheduler such as Cilk, any working thread may rightly be blamed for idleness: if that thread is not shedding parallel work, it is part of the cause of idleness. However, the same is not true for explicitly threaded programs. For example, if one thread is working but not holding a lock, then it is misleading for that thread to accept blame for threads contending for a lock. Consequently, evenly apportioning blame is not a sound strategy.

To rectify the problem of misappropriated blame, we redesigned the strategy to assign blame more precisely. In particular, we wish to apportion idleness deriving from lock contention only to threads that hold locks. We also wish to minimize the number of atomic increments that are required during critical sections.

We classify working threads W by one of two states (ignoring the case of over-subscription):

- W_l: working directly in a lock critical section
- W_{co}: working directly in a condition variable critical section or any other code (neither directly in a lock nor condition variable critical section)

Similarly, we classify idle threads I according to one of two states:

- I_l: idling at a (non-condition variable) lock
- I_c: idling at a condition variable (*i.e.*, waiting for a signal) or condition variable lock (*i.e.*, the thread has been signalled but is waiting to obtain the associated condition variable lock)

Given these observations, the most natural form of blaming is:

- Blame idleness I_l on workers in state W_l.
- Blame idleness I_c on workers in state W_{co} since any of the workers in state W_{co} could 1) signal the threads in state I_c or 2) unlock a condition variable lock.

Algorithm 1: An algorithm that, on sampling a working thread, computes that thread's share of the blame for the execution's idleness.

Assume: T, W, W_l and I_c are directly maintained.
Input: T, W, W_l and I_c

$W_l \Leftarrow \mathsf{max}(1, W_l)$ // $W_l \geq 1$
$I_c \Leftarrow \mathsf{max}(0, I_c)$ // $I_c \geq 0$
if *is working within lock* **then**
  let $I = (T - W)$ // $I \geq 0$
  let $I_l = \mathsf{max}(0, I - I_c)$ // $I_l \geq 0$
  return I_l/W_l
else
  let $W_{co} = \mathsf{max}(1, W - W_l)$ // $W_{co} \geq 1$
  return I_c/W_{co}

3.2 Making it practical

Clearly, it is possible to use four global counters to compute the number of idle and worker threads in states I_l, I_c, W_l, and W_{co}. Unfortunately, these counters require frequent adjustment within critical sections. Because a key implementation concern is minimizing the overhead of the Pthreads overrides, it is important to refrain from lengthening critical sections. For example, it is less of a problem for the override to perform bookkeeping *before* calling the actual `pthread_spin_lock` routine as opposed to after this routine has returned and the lock acquired. Therefore, it is important to minimize the number of atomic increments during critical sections.

It is possible to reduce the number of frequently maintained counters. Given that $T = W + I$, we have

$$W = W_l + W_{co}$$

$$I = T - W = I_l + I_c$$

Consequently, to compute all necessary values it is possible to use T (which only changes on thread creation/destruction) along with only three frequently adjusted counters, *e.g.*, W, W_l and I_c. All other state can be thread-local. By directly maintaining the suggested subset of counters, only two counters need to be atomically adjusted within lock and condition variable critical sections.

Algorithm 1 shows how this scheme apportions idleness when a sample is fielded by a working thread. If the worker is in category W_l, it attributes one unit of work to its work metric and I_l/W_l units of idleness to its idleness metric. Otherwise the worker is in category W_{co} and it attributes I_c/W_{co} units of idleness to its idleness metric. The algorithm uses max to account for possible timing windows between the (multiple) atomic increments that occur during the overrides.

It is worth noting that there are complications with correctly maintaining the global counters. For example, because critical sections can be nested, a thread can move from state W_{co} to W_l and back, which means that correctly maintaining counters requires some care.

3.3 Evaluation

Unfortunately, we found that even our extension to more precisely attribute blame was ineffective for complex programs. There are two key problems.

The first problem is that contention to atomically increment or decrement the global counters can be a significant issue. By

using tuned primitives and by preventing false sharing with cache-block alignment, we managed to bring overhead to an acceptable 5% on a 16-core machine. Nevertheless, even though we managed to achieve respectable overhead, the prospect of 48 and 64-core systems — or massively multithreaded systems such as the Cray XMT — suggests that global counters are likely to be an important weakness. A monitoring scheme should not itself cause significant amounts of new contention.

The second problem is even more fundamental. Even assuming low-overhead monitoring, we found that the lock-contention blame of this approach was still spread too diffusely for complex applications. While the approach of Section 2 attributes blame to *victims*, this approach targets *suspects*. While it is an improvement to attribute the idleness of lock-waiting threads to lock-working threads, the results can be inaccurate if most of the idling threads are waiting on one critical lock. For similar reasons, it can be misleading to attribute the idleness of 'cond'-waiting threads to all other working threads, even though any one could in theory potentially signal the condition variable. Consequently, for complex programs, we found blame to be too diluted because it is accumulated by actively working threads that have no relation to a source of contention.

4. Communicating blame directly to lock-holders

4.1 Blame shifting: A distributed and precise strategy

To pinpoint the cause of lock contention in its context, while avoiding the problems we have encountered thus far, we developed a fully distributed scheme that we call *blame shifting* to communicate blame for contention directly to lock-holders. Because it uses a fully distributed strategy and only lightweight instrumentation of synchronization primitives, it incurs very low overhead.

The key idea is to use a lock as a communication channel for directing blame. Consider the case of spin locks where threads busy-wait while contending for a lock. While profiling an application using sampling, threads contending for locks will receive samples while idling. When a thread takes a sample while waiting for a lock, we use an atomic add to accumulate that idleness in a counter associated with the lock. Then, when a thread that possesses a lock releases it, that thread blames itself for all of the idleness that accumulated while it held the lock. To accept blame, when a thread releases a lock, it atomically swaps zero into the lock's associated idleness counter. If the result of the swap is a non-zero value, then other threads must have contended for that lock while the lock-holder was working. So, the thread holding the lock attributes that idleness to the context of its lock release operation.

Although one might desire to attribute idleness to the lock acquisition point, using the release point provides a key benefit. Typically, there are several points in an execution where certain lock acquisitions are uncontested. Consequently, there are likely to be many lock release points where it is not necessary to incur the cost of unwinding the call stack to attribute zero blame. In contrast, attributing idleness to a lock acquisition point would require eager unwinds since that context may never again exist. Moreover, if a lock is contested only a short time, then it is unlikely to have a sample of idleness attributed to it. To see this, note that whereas a thread may acquire 100s-of-thousands of locks per second, it is sufficient to use sample rates on the order of 100-1000 samples/second for most programs.

4.2 Blame shifting in action

To implement blame shifting, it is necessary 1) to have thread-local data to indicate when a thread is not working and 2) to create a shared piece of monitoring state for each lock. As the former has been discussed in prior schemes, we focus on the latter.

4.2.1 In-band *vs.* out-of-band state

The first question is how to create the shared monitoring state. There are two possibilities: within the existing lock structure (in-band) or outside of it (out-of-band).

An in-band approach requires storing additional information within the existing lock. In particular, blame shifting requires a shared idleness counter for each lock. In general, reinterpreting bits within a data structure to add an extra field is difficult and at the very least requires overriding every routine that might access that data. Pthread's spin locks are simply 32-bit integers, even on 64-bit platforms. An in-band approach requires unevenly dividing this space into two fields to have enough room for the idleness counter. It also requires that the idleness field *never* overflow. It is also worth observing that both fields will be accessed by different threads and will be the target of atomic operations, even though neither is the natural architectural word size.

A second option is to create a special library and include file to implement an extended representation for a lock that includes a counter for blame shifting. This approach suffers from the disadvantage that one would need to recompile the application to use the larger lock structure. Because one of our underlying goals is to develop techniques that can be used to monitor unmodified programs, we consider such an option an approach of last resort. Of course, one could modify a system's standard threading library to use the extended representation for a lock; however, such an approach would not be portable.

A third approach is to allocate additional state associated with a lock in out-of-band data. A benefit of this approach over the in-band approach is that it is a more flexible solution; for example, additional monitoring state can easily be added. We implemented this approach.

4.2.2 Allocating out-of-band state

We now consider when to allocate this additional out-of-band state. At first glance, it might appear straightforward to allocate the out-of-band state when a lock is initialized with `pthread_{mutex, spin}_init`. This would be attractive since one could assume a race-free context. However, this approach is fraught with difficulty. First, while it is possible to override every instance of a Pthreads call, some of these overrides may occur in contexts in which a profiler cannot manage the out-of-band state. For example, Pthreads locks are often used very early during execution within `glibc` and during initialization of shared libraries and static constructors.

Second, supporting out-of-band lock state requires managing dynamic allocation and deallocation of state instances. In many programs, components of dynamic data structures are decorated with locks (*e.g.*, nodes in a tree). In such cases, a lock is destroyed when a node is freed; thus, managing the destruction of lock state is an essential part of an overall strategy for dynamic allocation. This shows that allocating out-of-band state for monitoring locks at the time of lock initialization requires the ability to dynamically allocate lock state and manage a per-thread free list[4] to which lock states could be appended when they are no longer needed. (Similarly, locks may be used after the application exits and monitoring tool shuts down but before the process has completely retired.) Providing both of these capabilities very early in an execution before the profiler is initialized is problematic.

Therefore, the shared lock state must in general be created on demand, *i.e.*, when the performance tool first sees an attempt at locking (which may be different than the first attempt at locking). This implies the state is created in a context where other lock operations might be executed concurrently.

[4] Using a per-thread free list avoids contention for the free list.

4.2.3 Accessing out-of-band state

On each call to a Pthreads locking routine, it is necessary to obtain the associated out-of-band state. There are two possibilities for accessing this data. The first option is to replace the contents of the lock itself with a pointer to a *monitored lock*. The second option is to write a function to quickly map between a pointer to a lock (which is unique) and its associated monitoring state.

The primary advantage of the first scheme is that finding a monitored lock can be an extremely fast constant-time operation. The primary disadvantage is that, because a performance tool might not see a lock's initialization, a native lock must be converted to a monitored lock within a race-sensitive context. For example, one thread may attempt to convert a lock into a monitored lock while that lock is currently held by a second thread and while a third thread is attempting to acquire that same lock. This implies that there must a concurrency protocol between the locking routines and the conversion routine.

The second option requires a data structure that supports both fast lookups and high concurrency. Because complex applications have a high rate of lock acquisitions, it is necessary to eschew coarse-grain locking. One potentially easy way to support high concurrency at the expense of extra memory is to make per-thread lookups faster by using an additional per-thread mapping data structure such as a splay tree. In other words, many lookups benefit from thread-local caches.

We initially tried the second approach because of its easier implementation. However, even using a local-global lookup to reduce contention on a centralized data structure — a balanced tree which itself used a sophisticated reader-writer lock — we were not satisfied with the resulting profiler overhead for programs that performed a high rate of lock acquisitions. Consequently, we developed protocols to support installing and managing monitored locks.

4.3 Dual-representation locks

To support fast accesses to shared lock state and to sidestep a difficult refactoring of profiler initialization to enable out-of-band monitored lock states to be used very early during execution, we opted to use a dual representation for locks. In prior work, Bacon *et al.* used a dual representation for object locks in Java [3], though for different reasons. We discuss this in more detail in Section 6.

Before profiler initialization, a lock is simply represented by a (32-bit) `pthread_spinlock_t`. Lock operations that occur before profiler initialization use this native lock representation. Once the profiler state is initialized, any lock, trylock, or unlock operation converts the native lock, in demand-driven fashion, to point to a monitored lock. The monitored lock includes the extra state needed to attribute contention. Once a lock has been converted into a monitored lock, it will remain a monitored lock until it is destroyed.[5] On each subsequent lock operation, the representation is examined, the monitored lock is obtained, and the operation proceeds using the monitored representation.

After profiler initialization, all lock, trylock, or unlock operations request a native lock's monitored lock by calling demand_mon_lock, shown in Algorithm 2. If the lock already represents a monitored lock, the routine simply accesses the associated monitored lock by reinterpreting the bits of the native lock. If a native lock is not yet a monitored lock, then the routine initiates a protocol for converting the native lock (of type `pthread_spinlock_t`) into a monitored lock. The protocol first allocates a new monitored lock

[5] Bacon *et al.* use an analogous approach for Java locks. Once they inflate a Java lock to a "fat" out-of-band representation, the lock remains inflated for its remaining life.

Algorithm 2: The protocol for converting a native lock into an out-of-band lock in demand-driven fashion.

```
1  typedef struct mon_lock { // a monitored lock
2    pthread_spinlock_t lock; // typedef'd as ''volatile int''
3    long idleness;
4  } mon_lock_t;
5  mon_lock_t* demand_mon_lock(pthread_spinlock_t* lock) {
6    if (!is_mon_lock(*lock)) {
7      mon_lock_t* mlock = alloc_mon_lock();
8      int newVal = make_mon_lock_ptr(mlock);
9      bool didSwap = false;
10     while (true) {
11       int curVal = *lock;
12       if (is_mon_lock(curVal)) break;
13       mlock->lock = curVal;
14       didSwap = (CAS(lock, curVal, newVal) == curVal);
15       if (didSwap) break;
16     }
17     if (!didSwap) free_mon_lock(mlock);
18   }
19   return get_mon_lock(*lock);
20 }
```

and computes a 'pointer' to install in the native lock.[6] Then, it enters the compare-and-swap (CAS) loop beginning on line 10. The loop obtains the current value of `lock` and ensures that since the test on line 6, `lock` is *still* a native lock. In that case, the protocol initializes a monitored lock with `lock`'s current value and attempts to atomically install a pointer to the monitored lock with the CAS on line 14. The loop exits when the CAS succeeds or some other thread converts the lock. If the latter occurs, the newly allocated monitored lock is reclaimed by placing it on a thread-local free list.

Algorithms 3–5 show the lock, trylock, and unlock protocols we use on these dual-representation locks. The algorithms are optimized for the typical case: a `pthread_spinlock_t` contains a pointer to a monitored lock.

The lock operation shown in Algorithm 3 works as follows. First it tests to see if the native lock has been overlaid with a pointer to a monitored lock state (line 4). If so, it extracts the pointer and then attempts to acquire the lock with a simple test-and-test-and-set protocol. While the lock word of the monitored lock is found to be in the LOCKED state, it continues to spin (line 9). When this condition is no longer true, some other thread must have set the lock word to its UNLOCKED state. A swap operation is used to atomically set the value of the lock word to LOCKED and recover its prior value. If the lock was UNLOCKED when the swap occurred, the lock acquisition is complete and the protocol returns. Otherwise, another thread acquired the lock. In that case, the protocol returns to the spin-wait loop where it again delays until the lock word is no longer LOCKED.

If a lock operation initially finds that `lock` does not point to a monitored lock, it enters a protocol to acquire the lock using the native representation. As with acquisition of a monitored lock, the protocol enters a loop that spin-waits for the lock representation to

[6] A `pthread_spinlock_t` is 32 bits, even for 64-bit programs. In a program running in 64-bit mode, this is not long enough to contain a full pointer. To address this problem, we allocate a segment for locks. We represent a lock pointer in a `pthread_spinlock_t` as an offset from a base address for the segment of monitored locks. For simplicity, in the rest of the paper we omit the quotation marks around 'pointer'.

Algorithm 3: Lock a dual-representation lock.

```
1  const int UNLOCKED = 1, LOCKED = 0;
2  int pthread_spin_lock(pthread_spinlock_t* lock) {
3    while (true) {
4      if (is_mon_lock(*lock)) {
5        // acquire a monitored lock
6        mon_lock_t* mlock = get_mon_lock(*lock);
7        lock = &mlock->lock;
8        while (true) {
9          while (*lock == LOCKED);
10         if (swap(lock, LOCKED) == UNLOCKED)
11           return 0; // success
12       }
13     }
14     // acquire a native unmonitored lock
15     while (*lock == LOCKED);
16     if (CAS(lock, UNLOCKED, LOCKED) == UNLOCKED)
17       return 0; // success
18   }
19   return 1; // failure
20 }
```

Algorithm 4: Trylock on a dual-representation lock.

```
1  int pthread_spin_trylock(pthread_spinlock_t* lock) {
2    while (true) {
3      if (is_mon_lock(*lock)) {
4        // trylock a monitored lock
5        mon_lock_t* mlock = get_mon_lock(*lock);
6        lock = &mlock->lock;
7        int prev = swap(lock, LOCKED);
8        return ((prev == UNLOCKED) ?
9          0 /* success */ : 1 /* failure */);
10     }
11     // trylock a native unmonitored lock
12     int prev = CAS(lock, UNLOCKED, LOCKED);
13     if (prev == UNLOCKED)
14       return 0; // success
15     else if (prev == LOCKED)
16       return 1; // failure
17   }
18 }
```

Algorithm 5: Unlock a dual-representation lock.

```
1  int pthread_spin_unlock(pthread_spinlock_t* lock) {
2    while (true) {
3      int lockval = *lock;
4      if (is_mon_lock(lockval)) {
5        // release a monitored lock
6        mon_lock_t* mlock = get_mon_lock(lockval);
7        mlock->lock = UNLOCKED;
8        return 0; // success
9      }
10     // release a native unmonitored lock
11     if (CAS(lock, LOCKED, UNLOCKED) == LOCKED)
12       return 0; // success
13   }
14   return 1; // failure
15 }
```

no longer be in the LOCKED state (line 15). When attempting to acquire an unmonitored lock, there are two conditions that might cause one to exit this spin-wait: another thread may have set the lock word to unlocked, or the profiler may have been initialized and another thread may have exchanged the lock word representation to point to a monitored lock. If the lock is available and in the UNLOCKED state, the subsequent compare-and-swap (CAS) operation will find it in the UNLOCKED state, set it to LOCKED, and return that it was in the UNLOCKED state. At this point the protocol will terminate after successfully acquiring the lock using the native representation. It is noteworthy that at this point in the protocol, it is necessary to use a CAS rather than a swap as used in the protocol for monitored locks. The reason is simple: the representation may have changed since we last inspected the lock word. If the lock word has been promoted to a pointer, one cannot obliviously overwrite it with LOCKED using a swap; instead, we conditionally overwrite it only if it is a native lock word in the UNLOCKED state. If the CAS fails, we return to the top of the outermost loop, check if the representation has changed, and execute the appropriate branch of the protocol to repeat the attempt to acquire the lock. An important feature of the protocol is that both the spin-wait and the CAS for the unmonitored lock representation can tolerate the representation being asynchronously switched to its monitored form. That would not be the case if line 15 read `while (*lock != UNLOCKED)` or line 16 used swap rather than CAS.

The trylock operation shown in Algorithm 4 similarly is designed to cope with our dual representation. If the lock word points to a monitored lock, it extracts the pointer and then attempts to acquire the lock with simple swap (line 7). Depending upon whether swap returns UNLOCKED, trylock succeeds or fails. Since a lock will never revert from a monitored lock pointer to a native representation until the lock is destroyed, if a lock is found to be using a monitored representation, it is safe to acquire it using a swap. If initially the lock word is not a pointer to out-of-band state, trylock attempts to acquire the lock in native form. In this case, the protocol uses a CAS operation (line 12) since the lock word may asynchronously change to a monitored lock pointer. If the lock word is still using the native representation (*i.e.*, with value LOCKED or UNLOCKED), the trylock returns immediately with the appropriate result. If the representation was asynchronously converted to a monitored lock pointer, execution will continue at the top the `while` loop on line 2, enter the protocol to try to acquire a monitored lock, and complete in a few operations. Note that that although this protocol contains a `while` loop, the loop will execute at most two iterations, resulting in a fixed number of instructions and leaving the trylock protocol non-blocking.

While the use of CAS in these dual-representation protocols is potentially more costly than simply using a swap to try to acquire a native lock, or using a simple write to unlock, this will have little impact on the run time cost of the locking protocol. These CAS operations execute only before profiler initialization. Since profiler initialization happens relatively early, in the typical case, the expected additional cost of the dual-representation in these protocols is limited to testing the lock word for a monitored lock pointer and converting that pointer into an actual pointer to the monitored lock.

The unlock operation shown in Algorithm 5 is quite similar to trylock in its handling for the dual representation. If the lock is found to point to a monitored lock, it simply sets the monitored lock's lock word to UNLOCKED. Otherwise, it attempts to unlock

the lock by using a CAS (line 11) to update the lock word from LOCKED to UNLOCKED. If this fails, the lock must have been asynchronously converted to a monitored lock pointer. A second pass around the `while` loop (line 2) will release the monitored lock. Although this protocol contains a `while` loop, the loop will execute at most two iterations, resulting in a fixed number of instructions and leaving the unlock protocol non-blocking. (This algorithm assumes a correct locking discipline).

4.4 Blocking (sleep-waiting)

Recall that when Pthreads mutex locks sleep-wait, they receive no samples. To implement blame shifting for sleep-waiting, we used a sampling strategy similar to that in Section 2.2. That is, on every n^{th} blocking call we time the thread's idleness and store it in the associated monitored lock's idleness counter. If the idleness count is non-zero when a thread releases the lock, it gathers the calling context. In principle this strategy should also work for condition variable waiting, but we have not implemented it.

4.5 Hints for developers

Many subtle implementation issues arise when overriding various Pthreads library functions for profiling. For our profiling tools to be broadly applicable, each issue needs to be solved generically in a way that induces low run time overhead. In some cases, the nature of interactions between target programs, run time systems, and our profiler forced more complicated solutions than originally desired.

For instance, overriding `pthread_mutex_lock` and performing any non-trivial operation involves subtle complexities. Many operations in thread-safe run-time libraries, such as `malloc` or `dlsym`, directly or indirectly call `pthread_mutex_lock` in at least some circumstances. The former would commonly be used to allocate out-of-band memory for monitoring locks; the latter for preparing the override for `pthread_mutex_lock`. To allocate dynamic memory, we use `mmap`-ed regions. To prepare the `pthread_mutex_lock` override, we use the special symbol `__pthread_mutex_lock` exported in the Linux implementation of Pthreads.

Although only a subset of Pthreads functions need to be wrapped, care must be taken to prevent inconsistent versions. Problems of this sort come in two flavors. First, one might wrap a Pthreads function that sets values visible to other functions that are not wrapped. One must choose the set of functions to wrap carefully to ensure that all functions sharing data have a consistent notion of appropriate states. Second, intra-library calls have to see a consistent world. In particular, calls that use hidden interfaces within libraries that cannot be overridden must be handled.

Finally, most unwinders — including HPCTOOLKIT's — are not designed to be recursive. Since our strategy uses both sampling-based call path profiling and synchronous unwinds of the call stack at a lock release point, it is important to specify what happens if an asynchronous sampling trigger occurs during a synchronous unwind. The simplest way to prevent interference is to prevent asynchronous samples during any unwind.

5. Case Studies

To show the effectiveness of blame shifting, we describe our experience applying it to three multithreaded applications with interesting locking and scheduling patterns. Our goal is to provide evidence that our method yields insight into non-trivial codes. In doing this, we distinguish between *obtaining* and *applying* insight. This is an important distinction because given an understanding of lock contention that includes a quantitative measure of the problem (insight), one might either resolve the problem or determine that a resolution is too costly (different applications). Because of the effort that would be involved in resolving the problems we identify, these studies focus on obtaining and not applying insight.

All experiments were performed on a Dell M905 blade running CentOS 5.2 and with four quad-core AMD 2.2 GHz Opterons (8354) and 48 GB main memory.

5.1 MADNESS

The first application we consider is MADNESS [16], a quantum chemistry application that makes extensive use of locking. MADNESS is designed to scale well both in SMP environments and on petascale clusters with multicore nodes. We focus on SMP executions here, but note that node-based performance is also critical for efficient performance on petascale clusters. MADNESS uses its own dynamic work scheduler based on a centralized queue. Worker threads create tasks (futures), which are pushed the queue. As necessary, workers pop tasks from the queue to obtain work. Among other things, MADNESS uses locks to manage access to the queue.

To obtain a sense of MADNESS's scaling losses, we gathered elapsed time for 4 and 16-core executions using the same input (strong scaling, averaged over five runs). While a 4-core run completed in 1150 seconds, a 16-core run took 516 seconds, an improvement of only a factor of 2.2. MADNESS' authors were aware of scaling losses but were unsure of the precise cause. Ignoring architectural concerns such as memory bandwidth, an obvious suspect is lock contention from managing the centralized task queue. However, it is not at all easy to show this for two reasons. First, understanding the different sources of lock contention in MADNESS is difficult because of its complex structure. Futures are implemented with templates. Typically, locks are implicitly acquired automatically through object creation and destruction. Furthermore, most critical sections are not straight-line code but a chain of templated method calls, heavily optimized by the compiler. Second, any monitoring tool must manage locks very efficiently to have low overhead for MADNESS. During a single 16-core execution, MADNESS used 65M distinct locks, had a maximum of 340K live locks, and performed an average of 30K lock acquisitions per second per thread. Finally, it is worth noting that MADNESS's authors had already spent considerable time experimenting with different implementation parameters.

We used our blame shifting strategy to measure lock contention on a version of MADNESS using spin locks. We used a sampling period of 5 ms to yield an average sampling rate of 200 samples/second. Curiously, during profiling, the execution time actually slightly *decreased* from 516s to 508s (averaged over 5 runs with no significant variability). We are not sure of the precise reason but note that this is an anomaly. Typically, our profiling overhead is positive, but less than 5%.

Figure 1 presents one view of the aggregated results displayed by our presentation tool. The view has three main components. The navigation pane (lower left sub-pane) shows a top-down view of the calling context tree, zoomed to focus on a portion of one call path. The call path is actually a fusion of dynamic calling contexts and the static context information such as loops and inlined frames. The selected line in the navigation pane highlights an instance of `ThreadPool::add` whose corresponding source code is shown in the source pane (top sub-pane). Each entry in the navigation pane is associated with metric values in the metric pane to the right. Two metrics are visible: '% idleness (all/I)' and '% idleness (all/E)'. Both metrics represent idleness as a percentage of total effort (giving the '%' qualifier) and summed over all threads (yielding the 'all' qualifier). (Recall that effort is the sum of work and idleness.) The former metric shows *inclusive* ('I') values, or values that are inclusive of an entry's children. The latter shows *exclusive* ('E') values that exclude its children. In the metric columns, metric values are shown in scientific notation. Note that because these particular metrics are percentages, the values in scientific notation are actually percents. The values formatted as percentages on the right

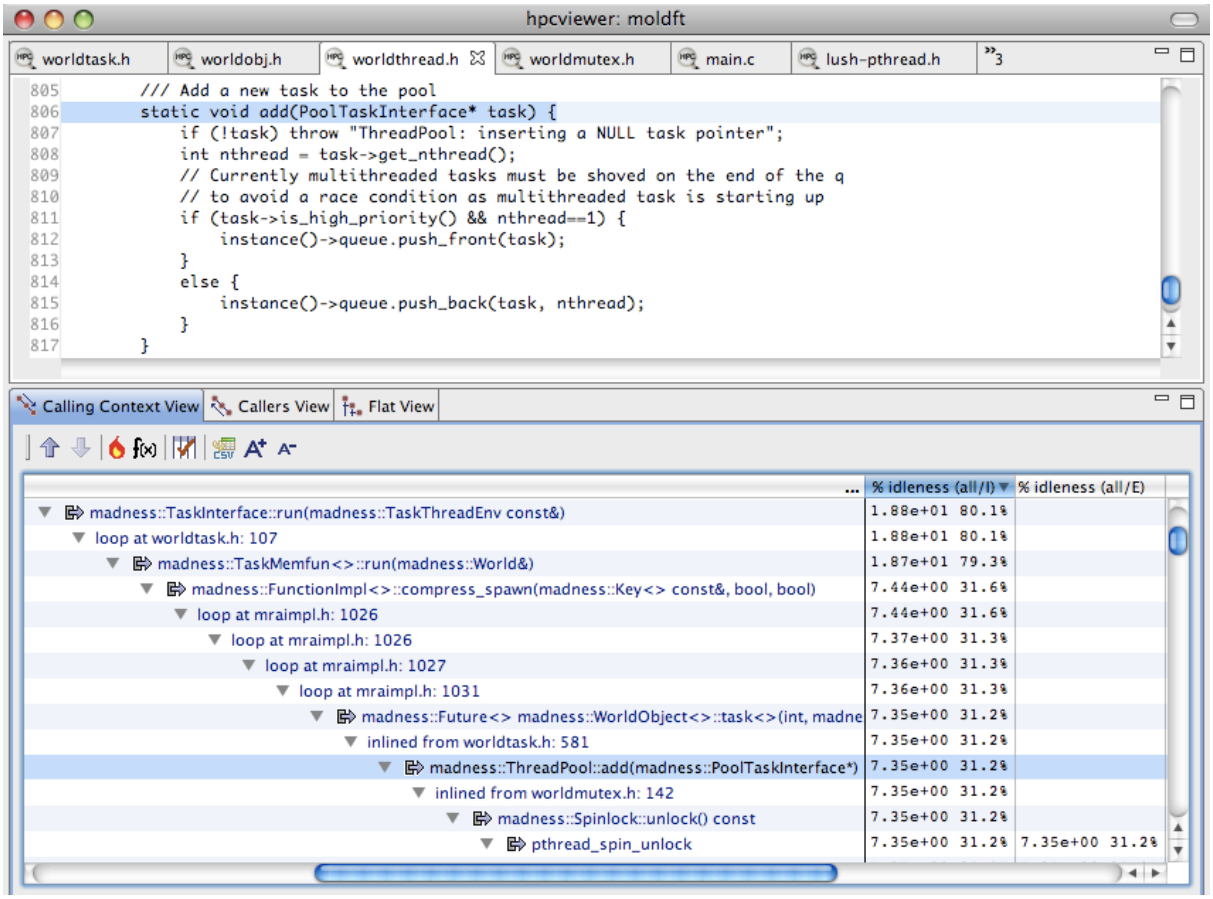

Figure 1. A Calling Context (top-down) view of MADNESS's `moldft`.

side of a column give an entry's proportion of the total idleness (as opposed to total effort).

The call path in the navigation pane is the hot call path with respect to the former metric and was expanded automatically. It is actually a fusion of dynamic calling contexts and static contextual information such as loops and inlined frames. The highlighted line in the navigation pane of Figure 1 indicates that 7.35% (scientific notation) of the total effort of the execution was spent in idleness at this context. Three lines below, we see the call to `pthread_spin_unlock`, exactly where blame shifting attributed the idleness due to lock contention. Within this call, both the inclusive and exclusive idleness metrics are identical, indicating that the call to `pthread_spin_unlock` accounts for all the idleness in this context.

This call path shows that there is lock contention associated with adding tasks to the centralized thread queue via `ThreadPool::add`. However, the remaining 68.8% of the idleness arises in other calling contexts. To avoid the need to search for other contexts in which there may be lock contention caused by `ThreadPool::add`, we turn to a bottom-up "Callers View" in Figure 2. If the top-down view looks 'down' the call chain, the *bottom-up* view looks 'up' to a procedure's callers. At the first level, the bottom-up view lists all the procedures in the program, rank-ordered according to the selected metric. Bottom-up metrics are computed by apportioning the total costs of a procedure to its calling context.

The first thing we observe is the very top line which gives aggregate values for the various metrics. (This line was not visible in Figure 1 because of scrolling.) We immediately see from the column labeled '% idleness (all/E)' that 23.5% of the execution's total effort consisted of lock contention. The column labeled 'idleness (all/E)' gives the absolute value of idleness (in microseconds): $1.57 \times 10^9 \mu s$. We should note that this value does not reflect all the idleness in the program. Because Pthreads does not provide a spin-based condition variable, MADNESS implements its own. In principle, we could instrument MADNESS itself. Since this is not the point of our work, our MADNESS results only measure regular lock contention and ignore any waiting at a condition variable critical section. However, we obtain an accurate measure of Pthreads spin lock contention.

When we automatically expand the hot path relative to the metric '% idleness (all/E)', we see something similar to the screen shot in Figure 2. This view shows how all the idleness attributed to `pthread_spin_unlock` is apportioned to its callers (in their context). Just above the selected line in the navigation pane is `ThreadPool::add`. Its associated idleness metrics show that it is responsible for 75.6% of the locking contention, accounting for 17.7% of the execution's total effort. This line not only confirms that adding tasks to a centralized queue is problematic, but quantifies its effect on idleness.

To see the effects of lock contention by context, we look up the call chain to the callers of `ThreadPool::add`. The selected line and its siblings (some of which are not shown) lists those callers (for this particular *callee* context). Since sibling entries in the navigation pane are sorted relative to their exclusive idleness (the selected metric), we can easily examine the handful of important ones. Doing this shows that most of the locking contention

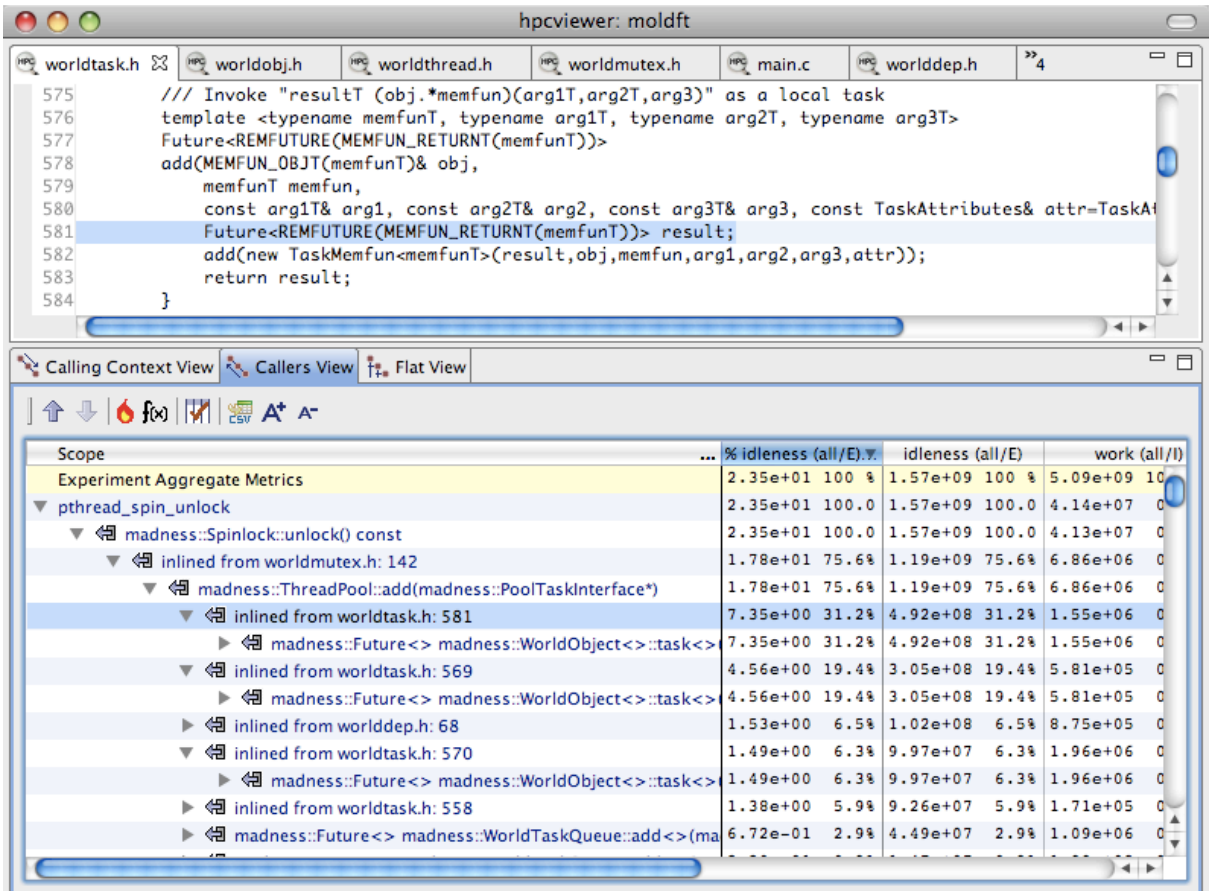

Figure 2. A Callers (bottom-up) view of MADNESS's `moldft`.

(67.5% of the total idleness) derives from creating `Futures`. The idleness costs are spread across distinct templates — *not* distinct instantiations — that manage `Futures` with different numbers of arguments. The selected line shows the templated add function for a `Future` with three arguments. An approach using distributed work queues and work stealing would likely significantly reduce lock contention.

Our original scaling experiment shows that we have not accounted for all scaling losses. There are at least two sources. First, the fact that memory bandwidth does not scale linearly with the number of cores is likely to be a factor. Second, besides missing idleness due to condition variable waiting, we cannot effectively monitor the non-idle overhead of creating and managing tasks. In prior work, we precisely computed overhead values for Cilk by modifying the Cilk compiler to distinguish between the user code and the parallel overhead [23]. While we have adopted this approach to identify the non-idle overhead of Pthreads routines, that overhead is negligible. The approach does not directly translate to MADNESS where there is no formal separation between the task management and the user code.

In hindsight, it is not surprising that a centralized queue protected by locks could introduce lock contention. However, it would be an error to conclude that these results are trivial. To see this, consider the question of how severe lock contention is on 8 cores. It turns out that the total lock contention on 8 cores is 1-2% because MADNESS' developers had optimized for this case. However, MADNESS' developers had no clear answers to questions like: How severe is lock contention for a particular execution? Do these executions fail to scale because of lock contention or some other reason? Is lock contention occurring primarily at the centralized queue or is it more evenly spread among other lock acquisitions? Our results help answer these questions.

5.2 UTS

The second case study is a Pthreads implementation of the Unbalanced Tree Search (UTS) benchmark [20]. UTS was designed to evaluate the performance and ease of programming parallel applications that require dynamic load balancing. UTS builds and searches trees where each vertex unpredictably either has no children or millions of descendants. The number of active vertexes varies between a few and tens-of-thousands during the execution (depending on the starting parameters and current depth).

UTS uses a work-stealing scheduler where each worker thread maintains a queue with two pieces, a local section that can be accessed without locks and a shared portion from which work can be stolen and which is protected by locks. A lock is acquired when work is moved from the local to the shared portion of a queue.

We profiled UTS and examined the resulting work and idleness metrics (microseconds) aggregated across all 16 threads. It was immediately apparent that although all cores were busy throughout the execution, they were only doing useful work about 40% of the time. With the idleness metric, we immediately pinpointed the source of idleness to contention for locks protecting the shared queues. About 72% of the idleness derived from contexts where new 'stealable' work was pushed onto the shared queues. Almost all of the remaining idleness (27.5%) was attributed to successful

steals of work by otherwise idle threads. Thus, a majority of this execution time was spent contending for the privilege of either providing or extracting work.

5.3 SSCA #2

The last case study is from the Scalable Synthetic Compact Application (SSCA) benchmark suite [9]. SSCA #2 was designed to be a hard-to-parallelize, compute-intensive analysis program that stresses memory access using integer and character operations.

We profiled an implementation of SSCA #2 using Pthreads written by Bader and Madduri [4]. Interestingly, idleness is very unevenly distributed across threads. In particular, 99.9% of the idleness of the first thread derives from a coarse-grained lock protecting an update to the graph. Having one lock per graph vertex rather than one graph-wide lock would reduce contention for that critical section and could greatly speed the initialization phase. The post-initialization compute kernels contained no significant sources of lock contention.

6. Related Work

Performance tools. Intel's Thread Profiler [5] (for Windows) has two ways to analyze multithreaded performance. First, it provides a measure of a routine's effective parallelism, a useful metric that is similar to Quartz [2] and the strategy of Hansen *et al.* [15]. Second, and more related to our work, it instruments synchronization objects with timers to further classify a thread's execution by its effects on other threads. Thread Profiler makes use of this information to 1) qualify a thread's execution and 2) to highlight synchronization objects that accumulate blocking time. To classify a thread's execution, Thread Profiler distinguishes between interaction effects such as cruise, impact and blocking time. *Cruise time* is time that a thread does not delay the next thread on the critical path while *impact time* is the opposite. If a thread on the critical path waits for some external event, it accumulates *blocking time*. While this is useful information, it requires substantial overhead to collect.

To highlight synchronization objects, Thread Profiler reports how much time was spent waiting for a particular object and the utilization of the system during that wait time [7]. It also shows the creation calling context of the synchronization object. If locks are statically allocated and have long lifetimes, this information can be very effective. However, additional information is needed if there is no direct line of sight from idleness at the lock to the source of contention. For example, only certain threads may be responsible for contention, locks may be dynamically created and destroyed (*e.g.*, linked data structures), or contention may be related to context. Our approach is superior to that of Thread Profiler in two ways. First, we 'blame' lock contention on the offending thread's context rather than aggregating wait time at a synchronization object; this directs an analyst to the source of the problem. Second, our approach is able to deliver this insight with very low monitoring overhead ($< 5\%$).

Several current tools detect lock contention in Java. IBM's Lock Analyzer for Java [17] computes a metric that reflects the number of delayed lock acquisitions as a percentage of total lock acquisitions. Sun's JConsole [8] helps identify contention by timing idle and by counting the number of delayed lock acquisitions. Like Intel's Thread Profiler, these tools attribute these metrics to locks themselves rather than to calling contexts and thus would fail to provide insight for an application like MADNESS.

Dual-representation locks. Bacon *et al.* use a dual representation for object locks in Java [3]. They use a 24-bit field in a Java object's header to implement a "thin lock" for objects that (a) are not subject to contention, (b) do not have wait, notify, or notifyAll operations performed upon them, and (c) are not locked to a nesting depth of more than 255. Objects that do not meet these criteria have their locks implemented as out-of-band "fat" locks. As with our scheme, once locks are converted to an out-of-band representation, they remain in that state. Bacon *et al.* avoid the need for a compare-and-swap in unlock because in their protocol, once a thread acquires a lock, no other thread may modify the lock word. In our approach, a lock may be changed to its out-of-band representation at any time. Without this, we would be unable to attribute contention to any lock that was acquired before profiling was initiated.

Contention managers for STM. In our work, we use auxiliary state associated with a lock to blame idleness resulting from contention for that lock on the lock holder and attribute the idleness to the calling context of the lock holder's unlock operation. Some contention managers for Software Transactional Memory (STM) use auxiliary state associated with transactional objects to notice and manage contention on the fly. For instance, the Eruption contention manager by Scherer and Scott [22] uses data associated with transactional objects not only to observe contention, but also to transfer priority from a blocked transaction to the transaction it is blocked behind. At an abstract level, both our profiler and the Eruption contention manager use state associated with synchronization objects to communicate information about contention between competing threads.

Hardware support for attributing stalls due to contention. The Alpha 21264's ProfileMe hardware support for instruction-based sampling [10] measures and quantifies the impact of contention for registers or execution units by measuring stalls while waiting for resources. While ProfileMe identifies contention and quantifies its impact, it attributes stall cycles to the victim of a stall rather than the instruction on which it is waiting. This strategy of attributing contention to waiting instructions is similar in effect to the strategy we describe in Section 2, which directly attributes contention to waiting threads.

7. Conclusions

Being able to quantify and attribute lock contention is important for understanding where a multithreaded program needs improvement.

We described three different approaches for quantifying lock contention that progressed from 1) attributing a thread's idleness to itself in the context in which it is idling (the victim); 2) then to the set of threads holding locks at the time (the suspects); and finally 3) to the thread holding the target lock (the perpetrator). Three underlying principles drove the development of our final blame shifting strategy. First, we strove to obtain a high degree of precision and detail in our measurements. Second, rather than sacrificing high overhead to obtain high precision, we developed extremely low overhead profiling methods. When using reasonable sampling rates (100-1000 samples/second), our overhead is typically $< 5\%$, even for an application that uses 65M distinct locks and an average of 30K lock acquisitions per second per thread. To prevent profiling itself from introducing serialization, we used a minimal amount of shared state and accessed it very rapidly. By using a sampling-based profiler that recovers call paths by unwinding a call stack, we were able to attribute idleness to its full static and dynamic context while maintaining extremely low overhead. We also used a form of sampling to amortize the cost of heavyweight instrumentation. Third, our aim was to develop a general method that enables tools to monitor unmodified programs. Doing this required solving subtle but complex problems such as how to maintain a dual-representation lock.

For future work, we would like to increase the precision of our results by recording the number of lock operations *within its calling context*. This would allow us to distinguish between a few highly

contested long waits and many moderately contested short waits. A low-overhead way of doing this is by collecting return counts [13].

Our profiler is based on the general principle of using shared state to communicate information about performance losses due to resource contention between competitors. While in this paper we apply this principle to attribute spin-waiting for a lock back to the calling context of the lock holder, we can imagine using variants of our strategy for other purposes. As one example, this same strategy could be used for reporting lock contention in multithreaded languages that provide locks such as Cilk. As another, in a lock-based software transactional memory system, transactions acquire locks associated with objects that they wish to modify transactionally. When another transaction needs an object that is already locked, a contention manager is invoked to decide which transaction to abort. Rather than just using using auxiliary object state to communicate information about contention and guide a contention manager's handling of competing transactional operations, our profiler could augment a transactional object with information that would enable us to attribute contention back to the transaction that holds an object lock and the calling context of the transaction.

Acknowledgments

Development of the HPCTOOLKIT performance tools is supported by the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25762.

We thank Robert Fowler for focusing our attention on MADNESS, Robert Harrison for helping us with his MADNESS code, and William Scherer for reminding us of Bacon's prior work on dual-representation locks and pointing out the similarity to STM contention managers. HPCTOOLKIT would not exist without the contributions of the other project members: Laksono Adhianto, Michael Fagan, Mark Krentel, and Gabriel Marin.

References

[1] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel Distributed Systems*, 1(1):6–16, 1990.

[2] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. *SIGMETRICS Perform. Eval. Rev.*, 18(1):115–125, 1990.

[3] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *Proc. of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, New York, NY, USA, 1998. ACM.

[4] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. *Lecture Notes in Computer Science*, 3769/2005:465–476, 2005.

[5] C. P. Breshears. Using Intel Thread Profiler for Win32 threads: Philosophy and theory. `http://software.intel.com/en-us/articles/using-intel-thread-profiler-for-win32-threads-philosophy-and-theory`, August 2007.

[6] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[7] S. Cepeda. Performance analysis and Intel Parallel Amplifier. `http://www.ddj.com/architect/217700473`, May 27, 2009.

[8] M. Chung. Monitoring and managing Java SE 6 platform applications. `http://java.sun.com/developer/technicalArticles/J2SE/monitoring`, August 2006.

[9] DARPA High Productivity Computing Program. Scalable Synthetic Compact Application benchmarks. `http://www.highproductivity.org/SSCABmks.htm`.

[10] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[11] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proc. of the International Symposium on Code Generation and Optimization*, pages 21–33, Washington, DC, USA, 2007. IEEE Computer Society.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998.

[13] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. of the 19th Annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.

[14] R. J. Hall. Call path profiling. In *Proc. of the 14th international Conference on Software engineering*, pages 296–306, New York, NY, USA, 1992. ACM Press.

[15] G. J. Hansen, C. A. Linthicum, and G. Brooks. Experience with a performance analyzer for multithreaded applications. In *Proc. of the 1990 ACM/IEEE Conference on Supercomputing*, pages 124–131, Washington, DC, USA, 1990. IEEE Computer Society.

[16] R. J. Harrison, G. I. Fann, T. Yanai, and G. Beylkin. Multiresolution quantum chemistry in multiwavelet bases. *Lecture Notes in Computer Science*, 2660/2003:103–110, 2003.

[17] IBM. IBM lock analyzer for Java. `http://www.alphaworks.ibm.com/tech/jla`.

[18] J. Larus and C. Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, 2008.

[19] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[20] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An unbalanced tree search benchmark. *Lecture Notes in Computer Science*, 4382/2007:235–250, 2007.

[21] G. F. Pfister and V. A. Norton. Hot-spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[22] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proc. of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[23] N. R. Tallent and J. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–240, New York, NY, USA, 2009. ACM.

[24] N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 441–452, New York, NY, USA, 2009. ACM.