Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures

Nicholas Hunt Paramjit Singh Sandhu Luis Ceze
University of Washington
{nhunt,paramsan,luisceze}@cs.washington.edu

Abstract

Accesses to shared data structures in multithreaded programs must be correctly synchronized to ensure data consistency and integrity. However, this synchronization between threads is a common source of performance problems in multithreaded applications. Lock-free data structures are an alternative to traditional synchronization methods that have potential for not only better performance and scalability, but better energy efficiency as well.

This paper presents a study of the relationship between the performance and energy consumption of various lock-free data structures based on the compare-and-swap primitive. We give a head-to-head comparison of lock-free and locking implementations of three data structures executing set of highly contentious workloads. We compare the execution time, peak power and total energy consumption of each and explain these results with the help of hardware performance counters. Our results show that under these workloads, the lock-free variants often perform better and use less energy then their traditional locking implementations.

1. Introduction

Data structures that are accessed by multiple threads of execution must be correctly synchronized to ensure data consistency and integrity. The most common technique for ensuring synchronized access relies on the principle of mutual exclusion among threads, protecting the data structure with a lock. Threads wishing to access shared data must first *acquire* the lock before access is allowed, and *release* it when they are finished; only a single thread can hold the lock at any time, and threads attempting to acquire the lock must wait until the lock is free. This technique serializes all access to shared data and thus tends to limit scalability with highly contentious workloads involving multiple threads of execution.

Alternatives to coarse-grained locking include more complex locking protocols (*e.g.,* a per-node lock instead of a single lock for an entire list); reader/writer locks, which give multiple threads read-only access to the data structure concurrently, but only allow a single thread to update the data structure at a time; and lock-free or wait-free data structures with more subtle implementations that do not rely on mutual exclusion for correctness.

This paper quantifies the differences in power, performance, and energy efficiency of lock-free and locking data structures. We examine a simple FIFO, a double-ended queue, and a sorted linked list. We chose to examine these three data structures due to both their prevalence in applications and their well-established lock-free implementations. We show that the relative performance and energy efficiency of the lock-free implementations are highly correlated, with better performance typically leading to lower total energy costs. However, we find no correlation between performance and power and thus lock-free data structures should not be seen as a silver bullet: programmers deciding between lock-free and locking implementations must evaluate the various trade-offs presented below to determine which implementation strategy is best for their situation.

1.1. Motivation

Prior work in lock-free and wait-free algorithms has focused mainly on the robustness properties and performance benefits, but few have considered the impact these lock-free structures have on the system's energy or power needs.

The interaction between performance, power, and energy is often a delicate balancing act. On one hand, faster algorithms tend to use more resources and thus typically increase the system's power requirements. However, how the new algorithm's energy requirements changes depends on the relative changes in performance and power and can be difficult to predict. Understanding this subtle interaction as it relates to the lock-free data structures examined in this paper can help guide software and library developers in their decision of which implementation to pursue, based on their energy and performance needs.

1.2. Contributions

This paper is the first, to the best of our knowledge, to characterize the energy and power utilization of lock-free data structures based on CAS primitives. Our results, which were obtained from measurements on actual hardware, show that a strong correlation exists between the performance of a data structure and its total energy consumption, and that under workloads with high contention, certain lock-free data structures offer significantly better performance at a lower energy cost. Further, our power measurements show that neither peak nor average power is correlated with the performance of the data structures. Finally, we present a characterization of execution including cache and branch prediction behavior as well as percentage of execution time spent executing system code.

1.3. Overview

The rest of this paper is organized as follows. Section 2 discusses the general background of lock-free data structures and some of the problems that arise during implementation. Section 3 summarizes our implementation of three different lock-free data structures. Section 4 presents the results of our energy and performance experiments and Section 5 discusses these results in greater detail. Section 6 discusses related work and Section 7 concludes.

2. Background

A relatively simple way to adapt a data structure for a multithreaded environment is to protect it with a mutually-exclusive lock: when any thread wishes to access the data structure, it must first *acquire* the lock, and after it finishes its task, it *releases* the lock, allowing other threads in turn to perform their own work. Any thread attempting to acquire the lock while it is currently held by another thread must wait until the lock is released before it can proceed. This policy ensures that only a single thread is modifying the data structure at a time and that no thread observes a partial update or some other form of inconsistent state.

Although simple, mutual exclusion can limit the scalability of a data structure. An alternative design is to use lock-free or wait-free algorithms that limit their use of synchronization constructs to primitive operations provided by the hardware, such as compare-and-swap (CAS) operations or software-supported transactional memory.

While addressing the scalability problem mentioned above, lock-free algorithms introduce implementation complexities of their own. For instance, the ABA problem can occur if two independent reads of a CAS target result in the same value, even though the state of the data structure has changed between the reads. Another difficulty is determining when it is safe to reclaim memory associated with the data structure, since another thread that is concurrently updating the data structure could still have a pointer to the data that needs to be freed. Both of these problems can be addressed by using hazard pointers as described in [8, 10, 11]. Details of this technique are omitted here for brevity.

3. Implementation

We implemented a coarse-grained locking and a lock-free version of a FIFO, a deque, and a sorted linked list. We also implemented a fined-grained locking version of the list for comparison. We expended reasonable effort minimizing the amount of work performed by the locking implementations in their critical sections.

Our lock-free implementations are based on algorithms described in prior work [11, 9, 2]. We adapted all implementations of the lock-free data structures to use hazard pointers for safe memory reclamation and ABA prevention, as described in [11].

3.1. FIFO Queue

The FIFO is implemented as a single-direction linked list, and maintains separate `head` and `tail` pointers for $O(1)$ insertion and deletion. Our lock-free implementation allows the `head` and `tail` pointers to be CAS'ed independently, and thus two threads can operate on opposite sides of the queue concurrently. Figure 1(a) shows our implementation diagrammatically, where each shaded boxes can be the target of a concurrent CAS operation. Allowing both ends of the FIFO to operated on concurrently should allow the lock-free implementation to have a theoretical 2x speedup compared to the locking implementation, which uses a single lock to protect the entire data structure.

3.2. Double-Ended Queue

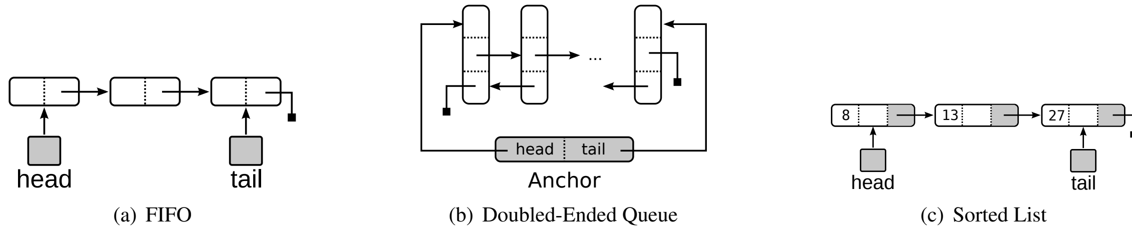
The double-ended queues (deques) are implemented as bidirectional lists, with separate `head` and `tail` pointers for $O(1)$ insertion and deletion from either end of the queue. Unlike the lock-free FIFO, however, the lock-free deque packs both the head and tail pointers into a single CAS target, known as the *anchor*. This means both pointers must be read or updated at the same time. Figure 1(b) shows diagrammatically how the lock-free version is implemented.

Because both pointers are manipulated as a single anchor value, the anchor becomes a point of serialization among all threads operating on the lock-free deque. Thus the deque imposes a serial order on all operations, limiting the available inherent parallelism in this data structure.

3.3. Sorted Linked List

The sorted linked list is implemented as a list of key/-value pairs. Each node has an integer key, and inserting values in the list ensures nodes remain in key-increasing order. In our lock-free implementation, each `next` pointer within the list can be CAS'ed independently from the others. Figure 1(c) shows the structure of the list.

We evaluate two different locking variants of the sorted



(a) FIFO (b) Doubled-Ended Queue (c) Sorted List

Figure 1. Layout of lock-free structures. Shaded values are CAS targets.

linked list. The first one protects the entire list with a single mutex, serializing all access to the data structure. The second version implements the fine-grained hand-over-hand locking protocol described in [3], using a separate lock for each node in the list; this recovers more parallelism by allowing multiple threads to update the data structure concurrently as long as they are updating disjoint parts of the list. We also implemented a version of the list that protects the list with a reader/writer lock. However, because this version behaved similarly to the coarse-grained mutex implementation, it is left out of the rest of this paper.

Of the three data structures examined in this paper, the lock-free list has the most potential for better performance, since all of the `next` pointers can be CAS'ed concurrently. Thus, if n threads are all updating the list at disjoint locations, all n CAS operations could be performed in parallel, providing a theoretical linear speedup with the number of threads.

4. Evaluation

In this section we present the results of our experiments, characterizing the performance and energy efficiency of the three data structures described above. Interpretation of these results is given in Section 5.

4.1. Methodology

All experiments were performed on an eight core Intel Nehalem machine with 2-way hyperthreading and 8 GB of RAM. To minimize noise in our performance and power measurements, all tests were ran with minimal background processes and were executed a minimum of 8 times each. The values reported below are the average over these runs (except for the peak power results shown in Section 4.5, which reports the maximum value seen across all runs).

Performance was evaluated by measuring the wall-clock execution time, as well as the percentage of time spent executing system code and user code, of each benchmark's region of interest (ROI). Wall-clock time was obtained with the `gettimeofday` system call, and the system/user times were obtained with the `times` system call. The ROI of a benchmark consists only of the operations being tested, and does not include the overhead of creating the data structure, populating it with initial data, displaying the results, etc. In each microbenchmark, the total workload size was kept fixed across each configuration: increasing the number of threads decreases the amount of work any single thread must perform, but the total work performed by all participating threads combined is kept constant.

The power of the test computer was measured with a Watts Up PRO meter connected directly to the test computer's power supply. Previous work [14] has been successful in correlating program performance and power when the meter was used in this fashion. The power meter samples the instantaneous power draw at a frequency of 1 Hz, and streams the timestamped samples to a second logging computer. Using the log produced, we computed the total energy consumed by the data structure while performing the workload by integrating the individual samples over time. Both computers synchronized their clocks via NTP throughout the experiments to ensure timestamp consistency in the logs. The clocks drifted no more than 20ms apart, although a difference of less than 5ms was typical.

Due to the low sample frequency of the power meter, we chose workload sizes that resulted in executions that were at least 30 seconds of execution time. For the characterization given in Section 4.5, we used the perfmon2 [5] performance monitoring interface to collect data from the processor's hardware performance counters.

4.2. Microbenchmarks

We implemented a set of microbenchmarks to simulate highly contentious workloads for these data structures, with very little application-level work performed for each data structure operation.

For the FIFO and deque, our microbenchmarks include enqueue-only workloads, where new values are added by all threads to the same end of the queue, and producer/consumer microbenchmarks where half of the threads add new nodes on one end, and the other half remove values from the opposite end. Because the deque implementation is symmetrical with respect to adding and removing elements from either end, only one direction is shown in the results.

The sorted linked-list has an insertion-only benchmark, consisting only of threads adding new values to the list: because the list is sorted, each insertion also requires the thread to search for the correct position in the list before the

insertion can occur. The keys for the newly inserted nodes were a randomly chosen 32-bit value, similar to what would result from a uniform hashing function.

Additionally, all data structures were also tested with a randomized microbenchmark, where all threads randomly chose the next operation to perform.

4.3. Performance Results

All graphs in this section show the execution times of the lock-free implementations normalized to those of the locking implementation: values greater than 1 represent runs where the lock-free implementation was *slower* than the locking version, and values less than 1 show cases where they were *faster*.

FIFO Queue. Figure 2(a) show the normalized runtimes of the lock-free FIFO microbenchmarks. Considering first the insertion-only benchmark, Figure 2(a) shows that for less than 16 threads, the lock-free implementation performs worse than the mutex-protected version; once the number of threads exceeds the hardware limit of 16, however, the lock-free implementation begins to perform better, taking about half the time of the baseline implementation at 128 threads. Although insertions into the lock-free FIFO have no more inherent parallelism than insertions into the locking implementation, the OS scheduler and other factors discussed in Section 4.5 eventually favor the lock-free version.

For the producer/consumer microbenchmark, as the number of threads increases, the execution time of the lock-free implementation approaches about 50% the total execution time of the locking version. This follows directly from the amount of parallelism inherent in the lock-free implementation: because the two ends of the FIFO can be operated on independently, two operations can be completed concurrently on opposite ends of the queue, whereas these same two operations would be serialized with the locking version of the FIFO.

The results of the random operation benchmark are very similar to the producer/consumer test. With a uniform random number generator choosing to either enqueue a value or deque a value, it is expected that roughly half of the threads will be performing each operation at any given time and thus this similarity is to be expected.

Double-Ended Queue. Figure 2(b) shows the runtime of the lock-free deque normalized to the baseline locking implementation. In general, this figure shows that the lock-free deque performs worse in all microbenchmarks up to 16 threads. Beyond 16 threads, the lock-free producer/consumer benchmark executed about 30% faster and the lock-free insertion-only microbenchmark executed about 10% faster with 128 threads. The randomized operation benchmark never performs better than the locking implementation, executing about 30-40% slower on average.

With no parallelism inherent in either implementation (the lock-free version serializes all updates on a packed anchor value as described in Section 3.2, and the locking implementation serializes all updates on the lock), there was no expectation that the deque should outperform the baseline. In fact, due to conflicting updates between threads, the lock-free implementation likely contains a large amount of redundant work due to contending updates. A possible explanation for why it outperforms the baseline at high thread counts is given later in Section 4.5.

Sorted Linked List. The runtimes for the lock-free list normalized to the locking implementation are shown in Figure 2(c). For comparison purposes, this graph also includes the normalized runtime of the fine-grained locking list described previously in Section 3.3. The list implementations each executed two benchmarks, one consisting of insertions only, and the other consisting of random choices between insertions, deletions and searches.

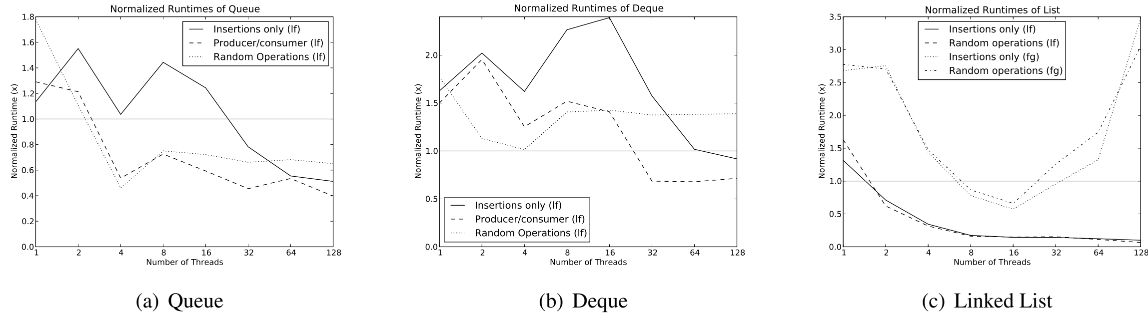
The lock-free implementation quickly outperforms the coarse-grained locking list, demonstrating almost linear speedup as the number of threads is increased. This level of parallelism is due to each node in the list acting as independent CAS target, allowing any number of threads to operate on the list concurrently, provided that each of the threads is operating at a distinct location in the list. Because the keys are uniformly distributed 32-bit values and number of operations performed by each thread vastly exceeds the number of threads, the probability that this occurs frequently is quite high.

Interestingly, despite having the same inherent parallelism of the lock-free implementation, the fine-grained locking implementation only performs better than the coarse grained locking protocol for the runs with 8 and 16 threads, and significantly worse for other thread counts. This is likely due to the implementation overhead of the locks used to protect the data structure. Because each node has a separate lock, a thread traversing the list must acquire and release significantly more locks than before: with n elements in the list, $O(n)$ locks must be acquired and released in the fine-grained implementation, compared to the $O(1)$ locks with coarse grained locking.

4.4. Energy

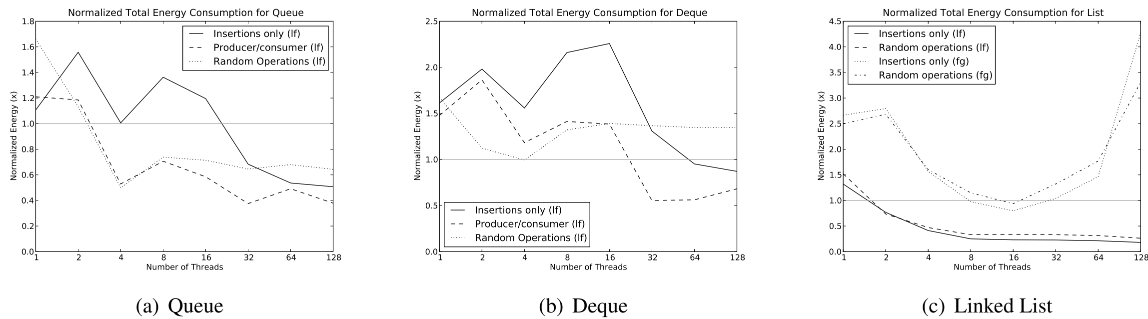
This section presents the results of our power tests. The values presented here were the average of at least 8 executions of each experiment. We used the Watts Up PRO power meter as described in Section 4.1 to sample the power draw of the test computer once a second and used these values to compute the total energy used during the ROI. Figures 3(a), 3(b), and 3(c) show total energy consumed by the FIFO, deque, and list while executing each of the microbenchmarks.

Qualitatively, these graphs are almost identical in shape to the execution time graphs shown previously in Sec-



(a) Queue (b) Deque (c) Linked List

Figure 2. Runtime of the lock-free implementations, normalized to the locking implementations. The linked-list graph shows both the lock-free (lf) implementation and the fine-grained locking (fg) implementation.



(a) Queue (b) Deque (c) Linked List

Figure 3. Total energy consumed by the lock-free implementations normalized to the locking implementations. The linked-list graph shows both the lock-free (lf) and the fine-grained locking (fg) implementations.

tion 4.3, and thus are not described in detail again. In fact, computing the Pearson correlation coefficient between benchmark execution time and total energy consumed yields a value that is always larger than 0.98, indicating a strong linear relationship between the two values: longer executions used more energy than shorter executions.

Despite attempting to put contending threads to sleep in the locking implementation, we did not observe reduced CPU utilization or lower energy consumption. Conversely, the speculative abort-and-retry methodology of the lock-free implementations did not require more energy, even though competing threads were never removed from the scheduling queues, thereby increasing CPU contention. Possible explanations for this non-intuitive result are presented in Section 4.5.

4.5. Characterization

This section provides the reader with a characterization of the dynamic behavior of each data structure implementation as they execute the various microbenchmarks described above. Its purpose is to provide an understanding of the other architecturally visible differences in the implementations and to provide a possible explanation of why the lock-free variants often perform better at high thread counts, even if no inherent parallelism exists within the data structure.

The characterization data is shown in Table 1. The data are shown in pairs: for each measurement, the column labeled as `MX` is the absolute value of the baseline, locking implementation, and the column labeled as `LF(x)` is the lock-free implementation's value normalized to the corresponding value in the `MX` column.

Instruction Counts. The dynamic instruction counts of the two implementations are shown in columns 4 and 5 of the table. Even though the total amount of application-level work is fixed across the different thread counts, the queue and deque execute more instructions as the number of threads is increased. One explanation for this is based on how the `pthreads` library implements the mutexes we used for synchronization. `pthreads` mutexes rely on futexes, the userspace mutex implementation in the Linux kernel. Each contended mutex acquire and release invokes the `sys_futex` system call, and this system call performs the requested operation, either a wait or a signal. However, the in-kernel futex state is itself protected with a spinlock, and

Struct	Bench	Num Th.	Instr. Cnt.		% Sys. Time		L1 Miss %		Br. Mispred.		Avg. power		Peak power	
			MX	LF(x)	MX	LF(x)	MX	LF(x)	MX	LF(x)	MX	LF(x)	MX	LF(x)
queue	Inserts	1	98 B	1.1	14%	0.84	< 1%	0.8	0.049%	0.95	171.0	1.0	178.1	1.0
		8	323 B	0.6	87%	0.01	11%	1.0	0.626%	3.81	191.2	1.0	206.0	1.0
		64	564 B	0.3	92%	0.19	23%	0.7	0.465%	3.63	168.5	1.0	192.8	0.9
	Pro/Con	1	99 B	1.1	13%	0.88	< 1%	0.8	0.049%	0.95	168.4	1.0	177.1	1.0
		8	362 B	0.4	86%	0.01	13%	0.8	0.632%	2.97	192.9	1.0	204.8	1.0
		64	520 B	0.3	92%	0.18	21%	0.8	0.387%	6.07	179.6	1.0	219.2	0.9
	Random	1	56 B	1.6	< 1%	< 0.01	< 1%	4.5	1.332%	0.49	168.3	1.0	176.0	1.0
		8	359 B	0.4	87%	0.01	12%	0.9	0.843%	3.03	194.0	1.0	206.0	1.0
		64	987 B	0.2	95%	0.01	39%	0.8	0.210%	18.54	216.7	1.0	219.9	1.0
deque	Inserts	1	98 B	1.5	12%	0.58	< 1%	0.6	0.048%	0.81	167.4	1.0	177.5	1.0
		8	368 B	1.3	87%	0.01	12%	0.7	0.632%	3.99	191.3	1.0	206.2	1.0
		64	637 B	0.5	92%	0.08	23%	0.7	0.464%	4.27	168.4	1.0	178.8	1.0
	Pro/Con	1	98 B	1.5	12%	0.58	< 1%	0.7	0.048%	0.80	167.3	1.0	175.2	1.0
		8	360 B	1.1	87%	< 0.01	12%	0.6	0.649%	4.16	193.2	1.0	206.0	1.0
		64	583 B	0.5	93%	0.12	26%	0.6	0.325%	7.40	186.2	0.9	218.4	0.9
	Random	1	66 B	1.8	< 1%	1.10	< 1%	36.4	1.268%	0.75	166.2	1.0	200.4	0.9
		8	434 B	1.0	81%	0.05	14%	0.6	1.138%	2.84	200.0	0.9	209.2	1.0
		64	984 B	0.8	93%	0.07	32%	1.0	0.334%	10.74	219.7	1.0	221.1	1.0
list	Inserts	1	96 B	2.2	< 1%	1.23	14%	1.1	0.003%	0.57	167.2	1.0	194.9	1.1
		8	80 B	2.2	1%	0.08	15%	1.2	0.013%	0.12	164.5	1.3	173.5	1.3
		64	81 B	2.2	1%	0.29	15%	0.9	0.018%	0.28	164.7	1.4	173.1	1.4
	Random	1	18 B	2.2	< 1%	< 0.01	15%	1.0	0.012%	0.41	166.5	1.0	172.9	1.3
		8	18 B	2.2	5%	0.05	15%	1.0	0.048%	0.12	165.6	1.4	166.9	1.4
		64	18 B	2.2	5%	0.31	15%	0.7	0.040%	0.21	163.6	1.4	169.1	1.5

Table 1. Characterization of the data structures. The MX columns are absolute results for the locking implementations and the LF(x) columns are for the lock-free implementations relative to the locking implementation.

as the number of threads is increased, contention on this internal spinlock is increased. Under these microbenchmarks, the additional spinning due to this growing contention is enough to significantly impact the total instruction count as shown. Thus, as the number of threads increases, the relative instruction count of the lock-free implementations decreases.

The instruction count for the sorted linked list behaves fundamentally different from those of the queue and deque. This is due to the difference in average cost of an update to these data structures. The queue and deque can each insert or remove an element with $O(1)$ work. The list, however, requires a traversal in order to maintain sorted order, and thus its updates require $O(n)$ work. This means the relative overhead of acquiring the mutex compared to the work performed while holding the mutex is significantly less than it is for the two previous data structures. As a result, both the absolute number of instructions executed by the locking list and the relative number of instructions performed by the lock-free list remains approximately constant as the number of threads is increased.

User vs. System. Columns 6 and 7 show the percentage of execution time spent executing system code. For all of the microbenchmarks (although much more so for the queue and deque than for the list), the percentage of time spent in the system increases with the number of threads. This is directly explained by the implementation of the mutexes discussed above: more time is spent spinning on the internal spinlock protecting futex state due to contention between threads trying to acquire or release the lock. The lock-free implementations, which do not use mutexes, spend much less time executing system code. The small percentage of system code executed by the lock-free implementations is due to the memory management performed by the benchmarks.

Further, because each contended operation on the lock results in a system call, the mutex protected data structures perform more context switches than the lock-free variants. Although not quantified in the table above, this additional overhead can also be a contributor to the poor performance of the locking data structures at high thread counts.

Cache Behavior. The L1 cache miss rate for the microbenchmarks is shown in columns 8 and 9 of the table. In general, the miss rates of the lock-free implementations grow with the number of threads at approximately the same rate as the locking implementations, although the lock-free implementations tended to have a lower miss rate in absolute value.

In two of the benchmarks, the relative miss rate of the lock-free implementation is rather large: 4.5x for the single-threaded random queue test, and 36.4x for the single-threaded random deque test. However, the difference between the locking and lock-free implementations in these cases were only 0.2% and 1.3%, respectively, and were not considered to be significant.

Branch Mispredictions. The branch misprediction rates are given in columns 12 and 13. Because branch mispredictions may involve costly pipeline flushes, high mispre-

diction rates can significantly hurt performance. The table shows that the locking benchmarks had high branch prediction accuracy, with the most having misprediction rates less than 0.5%, with a peak at 1.3%.

Compared to the baseline implementations, the lock-free queue and deque tend to do worse in this regard. Even though these benchmarks had slightly lower misprediction rates at a single thread, as the number of threads increased, the misprediction rates tended to grow as well. This is likely due to competing updates from multiple threads: if a thread observes a change in shared state after completing partial work, it must discard this work and try again. As the number of threads increases, the probability that a conflicting update will occur grows as well. Because the CAS operations are almost always used as the conditional expression in a branch statement, these nondeterministic branch conditions becoming more difficult to predict.

This trend is not evident in the linked list, however. There are two reasons for this. First, because updates from different threads are distributed across the list, the probability of conflicting updates is significantly lower than with the queue or deque, in which all threads are attempting to update only one or two different values. The second reason is that a nontrivial amount of execution time is spent traversing the linked list in a highly regular loop. As a result of these two factors, there is relatively little change in the misprediction rate as the number of threads grow.

Power. The average power and peak power for these benchmarks are shown in columns 12-13 and columns 14-15 respectively. Because the mutex implementation eventually puts threads waiting for a lock to sleep, the locking implementations should intuitively have a lower average power consumption than the lock-free implementations, which maintain high CPU utilization with speculative work rather than sleeping.

For the sorted linked-list, this intuition appears correct: the lock-free list imposes about a 40% overhead in the average power, and up to a 50% increase in peak power. For the lock-free queue and deque, however, the spinning in the futex implementation prevents these threads from being put to sleep, keeping CPU utilization high. Further, the critical sections for the queue and deque are so short that once a thread does go to sleep, the effect on the computer's power draw is likely minimal. Threads blocked on the list's mutex, however, are removed from the scheduling queues long enough to reduce the system's power draw, due to the significantly larger critical sections.

It was previously shown in Section 4.4 that the total energy consumption was linearly correlated with the execution time of the microbenchmarks. Based on the average and peak power values in this table, it is clear that the energy savings of the lock-free data structures is directly attributable to their shorter execution times. For the queue and deque, the locking and lock-free implementations required the same power resources, but the lock-free structures simply required them for less time. The lock-free list, which required significantly more power than the locking list, was able to compensate by executing for a significantly shorter duration.

5. Discussion

Power and Performance. In many domains there is a trade-off between power and performance: optimizations that lower power requirements will usually lower performance as well. Conversely, higher performing implementations usually come at the cost of higher power and energy due to the increased use of system resources. However, when comparing lock-free data structure implementations with their locking counterparts, we did not find such a clear trade-off.

The lock-free linked list behaved like a typical optimization in this regard. As the number of threads grew, the lock-free list saw a linear speedup; however, it need about 40% more power on average when compared with the baseline locking implementation. The lock-free FIFO, on the other hand, typically saw about a 2x performance improvement over the locking implementation, but did so with almost identical power requirements and thus required about half the energy to perform the same amount of work.

Futexes. In our study, we observed that the locking data structures were often unsuccessful putting threads to sleep due to contention on a kernel spinlock in the futex implementation. Although designing energy-aware locking primitives is beyond the scope of this work, we hypothesize that a different implementation of the mutex we used could change the results we observed: if threads were able to be removed from the scheduling queues more aggressively, it might make enough difference to reduce the power and energy costs of the locking implementations. However, due to the maturity of the Linux kernel, and in particular the spinlock implementation, we speculate that further optimization of the futex implementation would be non-trivial, and was not considered further in this study.

6. Related Work

Lock-free data structures have been an active area of research for over a decade [2, 9, 15, 1]. Most of this prior work, however, has focused on the correctness of the implementation or demonstrating improved performance or reliability compared to prior implementations: they do not generally consider the effect on the power and energy requirements of the computer running them. This work shows that even though power and energy were not first class design constraints in the development of these lock-free algorithms, these existing implementations can lead to a signifi-

cant reduction in the energy costs of the data structure.

Energy efficiency in general is also an active area of research, and now even more so than ever, due to the increasing growth of mobile computing and large data centers. Systems like those described by Merkel *et al.* [7] and Singh *et al.* [14] propose energy-aware scheduling techniques.

However, as many-core devices become increasingly popular, understanding the energy costs of synchronization in shared-memory multiprocessor environments is of growing importance important. Li *et al.* [6] examine the isolated case of reducing the energy consumed by an application due to barrier synchronization imbalance: by predicting the amount of time a thread will need to wait at a barrier, rather than simply spinning, the processor can transition to an appropriate low-power processor state to decrease amount of energy expended while the thread waits.

Moreshet *et al.* [13, 12] advocate hardware transactional memory as an energy efficient way of implementing lock-free data structures. Through simulation, Moreshet *et al.* show up to 90% reduction in energy is possible in some situations. However, because hardware transactional memory support is not currently available on commodity processors, this approach doesn't provide a solution today. Although software only implementations of transactional memory do exists today, Klein *et al.* [4] demonstrate that software implementations typically lead to energy *inefficiencies* when subjected to high contention workloads. The present paper focuses instead on the energy efficiency of lock-free data structures using atomic primitives readily available on commercial processors, rather than relying on hardware or software support for transactional memory.

7. Conclusions

With growing use of parallel and concurrent software in energy-constrained devices, it is becoming more important to understand the energy and power efficiency of synchronization primitives. In workloads with contended shared data, the choice of synchronization techniques can have a significant impact on both the performance and energy efficiency of applications. Traditional methods of protecting data structures with coarse-grained, mutual-exclusion locks are easy for programmers to understand and implement, but suffer from scalability issues due to serialization.

In this work, we have shown that lock-free data structures can not only provide significant performance improvements in many situations, but also that this increase in performance can improve the data structure's energy efficiency as well. Further, we've shown that the increase in performance does not necessarily come with an increase in peak power, an important observation in the context of low-power devices.

As application and library developers choose implementations for their data structures, it is becoming more important to consider factors other than just correctness and performance. Energy and power constrained devices such as cell-phones and embedded systems are becoming more pervasive and developers should choose implementations according to their power, energy and performance needs. For instance, because the lock-free linked list uses more power than the locking implementation, it may not be a viable alternative low-power situations. However, the lock-free FIFO, which uses the same power but performs better would be useful since performance and battery life could be extended. Thus, while lock-free data structures are not necessarily a silver bullet, its important to consider the various costs and benefits of lock-free alternatives when choosing implementations for data structures. This paper presented the types of information that system or library programmers can use when making this decision for their particular situation.

References

[1] D. Detlefs et al. Even better dcas-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, 2000.

[2] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC*, 2004.

[3] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[4] F. Klein et al. On the energy-efficiency of software transactional memory. In *SBCCI*, 2009.

[5] H. P. Labs. perfmon2: The hardware-based performance monitoring interface for linux. `http://perfmon2.sourceforge.net/`.

[6] J. Li, J. F. Martínez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *HPCA*, 2004.

[7] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *EuroSys*, 2006.

[8] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC*, 2002.

[9] M. M. Michael. Cas-based lock-free algorithm for shared deques. In *Euro-Par*, 2003.

[10] M. M. Michael. Aba prevention using single-word instructions. Technical Report 23089, Thomas J. Watson Research Center, 2004.

[11] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE TPDS*, 2004.

[12] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy-aware microprocessor synchronization: Transactional memory vs. locks.

[13] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *ISLPED*, 2005.

[14] K. Singh, M. Bhadauria, and S. A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Computer Architecture News*, 2009.

[15] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Götenborg University, 2004.