SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Software-based contention management for efficient compare-and-swap operations

Dave Dice[1], Danny Hendler[2] and Ilya Mirsky[2]

[1] *Oracle Labs*
[2] *Ben-Gurion University of the Negev and Telekom Innovation Laboratories*

SUMMARY

Many concurrent data-structure implementations – both blocking and non-blocking – use the well-known *compare-and-swap* (CAS) operation, supported in hardware by most modern multiprocessor architectures, for inter-thread synchronization.
A key weakness of the CAS operation is its performance in the presence of memory contention. When multiple threads concurrently attempt to apply CAS operations to the same shared variable, at most a single thread will succeed in changing the shared variable's value and the CAS operations of all other threads will fail. Moreover, significant degradation in performance occurs when variables manipulated by CAS become contention "hot spots", since failed CAS operations congest the interconnect and memory devices and slow down successful CAS operations.
In this work we study the following question: *can software-based contention management improve the efficiency of hardware-provided CAS operations?* In other words, can a software contention management layer, encapsulating invocations of hardware CAS instructions, improve the performance of CAS-based concurrent data-structures?
To address this question, we conduct what is, to the best of our knowledge, the first study on the impact of contention management algorithms on the efficiency of the CAS operation.
We implemented several Java classes, that extend Java's *AtomicReference* class, and encapsulate calls to the native CAS instruction with simple contention management mechanisms tuned for different hardware platforms. A key property of our algorithms is the support for an almost-transparent interchange with Java's AtomicReference objects, used in implementations of concurrent data structures. We evaluate the impact of these algorithms on both a synthetic micro-benchmark and on CAS-based concurrent implementations of widely-used data-structures such as stacks and queues.
Our performance evaluation establishes that lightweight software-based contention management support can greatly improve performance under medium and high contention levels while typically incurring only small overhead under low contention. In some cases, applying efficient contention management for CAS operations used by a simpler data-structure implementation yields better results than highly optimized implementations of the same data-structure that use native CAS operations directly.
Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Compare-and-swap, contention management, concurrent algorithms.

1. INTRODUCTION

Many key problems in shared-memory multiprocessors revolve around the coordination of access to shared resources and can be captured as *concurrent data structures* [1]: abstract data structures

Contract/grant sponsor: Partially supported by the Israel Science Foundation (grant number 1227/10) and by the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

Copyright © 0000 John Wiley & Sons, Ltd.

that are concurrently accessed by asynchronous threads. Efficient concurrent data structures are key to the scalability of applications on multiprocessor machines. Devising efficient and scalable concurrent algorithms for widely-used data structures such as counters (e.g., [2, 3]), queues (e.g.,[4, 5, 6, 7, 8, 9, 10, 11, 12], stacks (e.g.,[5, 7, 13]), pools (e.g.,[14, 15, 16]) and hash tables (e.g., [17, 18, 19, 20]), to name a few, is the focus of intense research.

Modern multiprocessors provide hardware support of atomic read-modify-write operations in order to facilitate inter-thread coordination and synchronization. The *compare-and-swap* (CAS) operation has become the synchronization primitive of choice for implementing concurrent data structures - both lock-based and nonblocking [21] - and is supported by hardware in most contemporary multiprocessor architectures [22, 23, 24]. The CAS operation takes three arguments: a memory address[1], an old value, and a new value. If the address stores the old value, it is replaced with the new value; otherwise it is unchanged. The success or failure of the operation is then reported back to the calling thread. CAS is widely available and used since its atomic semantics allow threads to read a shared variable, compute a new value which is a function of the value read, and write the new value back only if the shared variable was not changed in the interim by other, concurrent, threads. As proven in Herlihy's seminal paper [21], CAS can implement, together with reads and writes, any object in a wait-free manner.

A key weakness of the CAS operation, known to both researchers and practitioners of concurrent programming, is its performance in the presence of memory contention [25]. When multiple threads concurrently attempt to apply CAS operations to the same shared variable, typically at most a single thread will succeed in changing the shared variable's value and the CAS operations of all other threads will fail. Moreover, significant degradation in performance occurs when variables manipulated by CAS become contention "hot spots", since failed CAS operations congest the interconnect and memory devices and slow down successful CAS operations.

To illustrate this weakness of the CAS operation, Figure 1 shows the results of a simple test, conducted on an UltraSPARC T2+ (Niagara II) chip, comprising 8 cores, each multiplexing 8 hardware threads, in which a varying number of Java threads run for 5 seconds, repeatedly reading the same variable and then applying CAS operations attempting to change its value.[2] The number of successful CAS operations scales from 1 to 4 threads but then quickly deteriorates, eventually falling to about 16% of the single thread performance, less than 9% of the performance of 4 threads. As we show in Section 3, similar performance degradation occurs on Intel's Xeon and i7 platforms.

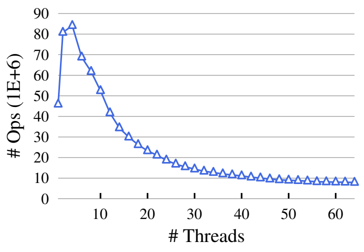


Figure 1. SPARC: Java's CAS

In this article, which extends our paper [26], we study the following question: *can software-based contention management improve the efficiency of hardware-provided CAS operations?* In other words, can a software contention management layer, encapsulating invocations of hardware CAS instructions, significantly improve the performance of CAS-based concurrent data-structures?

To address this question, we conduct what is, to the best of our knowledge, the first study on the impact of contention management algorithms on the efficiency of the CAS operation. We implemented several Java classes that extend Java's AtomicReference class, and encapsulate calls to direct CAS by a *contention management layer*. This design allows for an almost transparent plugging of our classes into existing data structures which make use of Java's AtomicReference. We then evaluated the impact of these algorithms on the Xeon, SPARC and i7 platforms by using both a synthetic micro-benchmark and CAS-based concurrent data-structure implementations of stacks and queues.

[1]In object-oriented languages such as Java, the memory address is encapsulated by the object on which the CAS operation is invoked and is therefore not explicitly passed to the interface to the CAS operation.
[2]We provide more details on this test in Section 3.

We note that the lock-freedom and wait-freedom progress properties aren't affected by our contention management algorithms since in all of them a thread only waits for a bounded period of time.

Our performance evaluation establishes that lightweight contention management support can significantly improve the performance of concurrent data-structure implementations as compared with direct use of Java's AtomicReference class. Our CAS contention management algorithms improve the throughput of the concurrent data-structure implementations we experimented with by a factor of up to 12 for medium and high contention levels, typically incurring only small overhead in low contention levels.

We also compared relatively simple data-structure implementations that use our CAS contention management classes with more complex implementations that employ data-structure specific optimizations. We have found that, in some cases, applying efficient contention management at the level of CAS operations, used by simpler and non-optimized data-structure implementations, yields better performance than that of highly optimized implementations of the same data-structure that use Java's AtomicReference objects directly.

Our results imply that encapsulating invocations of CAS by lightweight contention management algorithms is a simple and generic way of significantly improving the performance of concurrent objects. The code implementing our algorithms and benchmarks is publicly available [27].

1.1. Related work

The idea of employing contention management and, more specifically, backoff techniques to improve performance was widely studied in the context of *transactional memory* (TM) and lock implementations. Herlihy et al. [28] were the first to suggest the use of a contention management (CM) module for ensuring the progress of transactional memory implementations. Upon a collision, a transaction consults the CM module in order to determine whether it should abort the other transaction immediately or wait in order to allow the other transaction to commit. In the wake of this work, TM contention management became an active area of research (see, e.g., [28, 29]).

Anderson [30] investigated software spin-waiting mechanisms for effectively handling accesses of a contended lock based on an atomic instruction such as test-and-set. This line of research was pursued by several additional works (see, e.g., [30, 31, 32])

Backoff techniques are also used at the higher abstraction level of specific data structure implementations. For instance, Hendler et al. [13] presented the *elimination-backoff stack* that copes with high-contention by attempting to pair-up concurrent push and pop operations that "collide" on entries of a so-called *elimination array*. In addition, it employs an exponential-backoff scheme after a CAS failure. Flat combining [7] protects a sequential data-structure by a single lock and associates with it a dynamic publication list where threads publish their operations to the data-structure. Threads that succeed in capturing the lock apply operations on behalf of other threads. Flat combining manages to reduce contention and synchronization overheads for low and medium concurrency levels.

Contention-management at the data-structure level adds complexity to the design of the data-structure and requires careful per-data structure tuning. Our approach, of adding contention management mechanisms at the CAS instruction level, provides a simple and generic solution, in which tuning can be done *per architecture* rather than per implementation.

The rest of this paper is organized as follows. We describe the contention management algorithms we implemented in Section 2. We report on our experimental evaluation in Section 3. We conclude the paper in Section 4 with a short discussion of our results.

2. CONTENTION MANAGEMENT ALGORITHMS

In this section, we describe the Java CAS contention management algorithms that we implemented and evaluated. These algorithms are implemented as classes that extend the *AtomicReference* class of the *java.util.concurrent.atomic* package. Each instance of these classes operates on a specific location in memory and implements the *read* and *CAS* methods.[3]

In some of our algorithms, threads need to access a per-thread state associated with the object. For example, a thread may keep record of the number of CAS failures it incurred on the object in the past in order to determine how to proceed if it fails again. Such information is stored as an array of per-thread structures. To access this information, threads call a *registerThread* method on the object to obtain an index of an array entry. This thread index is referred to as *TInd* in the pseudo-code. After registering, a thread may call a *deregisterThread* method on the object to indicate that it is no longer interested in accessing this object and that its entry in this object array may be allocated to another thread.[4]

Technically, a thread's *TInd* index is stored as a thread local variable, using the services of Java's *ThreadLocal* class. The *TInd* index may be retrieved within the CAS contention management method implementation. However, in some cases it might be more efficient to retrieve this index at a higher level (for instance, when CAS is called in a loop until it is successful) and to pass it as an argument to the methods of the CAS contention management object.

2.1. *The ConstantBackoffCAS Algorithm*

Algorithm 1 presents the *ConstantBackoffCAS* class, which employs the simplest contention management algorithm that we implemented. No per-thread state is required for this algorithm. The *read* operation simply delegates to the *get* method of the AtomicReference object to return the current value of the reference (line 3). The *CAS* operation invokes the *compareAndSet* method on the AtomicReference superclass, passing to it the *old* and *new* operands (line 6). The *CAS* operation returns *true* in line 9 if the native CAS succeeded. If the native CAS failed, then the thread busy-waits for a platform-dependent period of time, after which the *CAS* operation returns (lines 7–8).

Algorithm 1: `ConstBackoffCAS`

```
1  public class ConstBackoffCAS<V>
   extends AtomicReference<V>
2  public V read() {
3      return get()
4  }
5  public boolean CAS(V old, V new) {
6      if ¬compareAndSet(old,new) then
7          wait(WAITING_TIME)
8          return false
9      else return true
10
11 }
```

2.2. *The TimeSliceCAS Algorithm*

Algorithm 2 presents the *TimeSliceCAS* class, which implements a time-division contention-management algorithm that, under high contention, assigns different time-slices to different threads. Each instance of the class has access to a field $regN$ which stores the number of threads that are currently registered at the object.

The *read* operation simply delegates to the *get* method of the AtomicReference class (line 14). The *CAS* operation invokes the *compareAndSet* method on the AtomicReference superclass (line 16). If the CAS is successful, the method returns *true* (line 16).

If the CAS fails and the number of registered threads exceeds a platform-dependent level *CONC* (line 17), then the algorithm attempts to limit the level of concurrency (that is, the number of threads concurrently attempting CAS on the object) at any given time to at most *CONC*. This is done as follows. The thread picks a random integer slice number in $\{1, \ldots, \lceil regN/CONC \rceil\}$ (line 19). The length of each time-slice is set to 2^{SLICE} nanoseconds, where *SLICE* is a platform-dependent integer. The thread waits until its next time-slice begins and then returns false (lines 20–24).

[3]None of the methods of AtomicReference are overridden by our implementation.
[4]An alternative design is to have a global registration/deregistration mechanism so that the *TInd* index may be used by a thread for accessing several CAS contention-management objects.

Algorithm 2: `TimeSliceCAS`

```
12 public class TimeSliceCAS<V> extends
   AtomicReference<V>
13 private int regN
14 public V read() { return get() }
15 public boolean CAS(V old, V new) {
16     if compareAndSet(old,new) then return true
17     if regN > CONC then
18         int totalSlices = ⌈regN/CONC⌉
19         int sliceNum = Random.nextInt(totalSlices)
20         repeat
21             currentSlice = (System.nanoTime() >>
               SLICE) % totalSlices
22         until sliceNum = currentSlice
23     end
24     return false
25 }
```

Algorithm 3: `ExpBackoffCAS`

```
26 public class ExpBackoffCAS<V> extends
   AtomicReference<V>
27 private int [] failures = new int [MAX_THREADS]
28 public V read() { return get() }
29 public boolean CAS(V old, V new) {
30     if compareAndSet(old,new) then
31         if failures[TInd] > 0 then failures[TInd]−−
32
33         return true
34     else
35         int f = failures[TInd]++
36         if f > EXP_THRESHOLD then
37             wait(2^{min(c·f,m)})
38         end
39         return false
40     end
41 }
```

2.3. *The ExpBackoffCAS Algorithm*

Algorithm 3 presents the *ExpBackoffCAS* class, which implements an exponential backoff contention management algorithm. Each instance of this class has a *failures* array, each entry of which – initialized to 0 – stores simple per-registered thread statistics about the history of successes and failures of past CAS operations to this object (line 27). The *read* operation simply delegates to the *get* method of the AtomicReference class (line 28).

The *CAS* operation invokes the *compareAndSet* method on the AtomicReference superclass (line 30). If the CAS is successful, then the *CAS* operation returns *true* (line 33).

If the CAS fails, then the thread's entry in the *failures* array is incremented and if its value f is larger than a platform-dependent threshold, the thread waits for a period of time proportional to $2^{min(c \cdot f, m)}$ where c and m are platform-dependent integer algorithm parameters (lines 35–36).

2.4. *The MCS-CAS Algorithm*

With the MCS-CAS algorithm, threads may apply their operations in either *low-contention mode* or *high-contention mode*. Initially, a thread starts operating in low-contention mode, in which it essentially delegates read and CAS operations to the respective methods of the AtomicReference class. When a thread incurs *CONTENTION_THRESHOLD* (a platform-dependent constant) consecutive CAS failures on a specific memory location, it reverts to operating in high-contention mode *when accessing this location*.

In high-contention mode, threads that apply CAS operations to the same memory location attempt to serialize their operations by forming a queue determining the order in which their read and CAS operations-pairs will be performed. Threads wait for a bounded period of time within their read operation and proceed to perform the read (and later on the CAS) once the thread that precedes them in the queue (if any) completes its CAS operation.

MCS-CAS implements a variation of the Mellor-Crummey and Scott (MCS) lock algorithm [31]. Since we would like to maintain the nonblocking semantics of the CAS operation, a thread t awaits its queue predecessor (if any) for at most a platform-dependent period of time. If this waiting time expires, t proceeds with the read operation without further waiting. If all threads operate in high-contention mode w.r.t. memory location m (and assuming the waiting-time is sufficiently long), then all CAS operations to m will succeed, since each thread may read m and later apply its CAS to m without interruption. In practice, however, threads may apply operations to m concurrently in both low- and high-contention modes and failures may result. After successfully performing a platform-dependent number of CAS operations in high-contention mode, a thread reverts to operating in low-contention mode.

If a thread needs to apply a read that is not followed by a CAS, then it may directly apply the *get* method of the AtomicReference super-class as this method is not overridden by the MCS-CAS class. There may be situations, however, in which it is not known in advance whether a read will be followed by a CAS and this depends on the value returned by the read. Such scenarios will not compromise the correctness and non-blocking progress of MCS-CAS, but may have adverse effect on performance. This comment applies also to the array based algorithm described in Section 2.5. The full pseudo-code of the MCS-CAS algorithm and its description is provided in Appendix A.

2.5. The ArrayBasedCAS Algorithm

The array-based algorithm uses an array-based signaling mechanism, in which a *lock owner* scans the array for the next entry, on which another thread is waiting for a permission to proceed, and signals it. Also in this algorithm, waiting-times are bounded.

There are two key differences between how *MCS-CAS* and *ArrayBasedCAS* attempt to serialize read and CAS operations-pairs to a memory location under high contention. First, whereas in MCS-CAS a thread signals its successor after completing a *single* read/CAS operations-pair, with the array-based algorithm a thread performs a multiple, platform-dependent, number of such operation-pairs before signaling other waiting threads.

A second difference is that, whereas MCS-CAS forms a dynamic queue in which a thread signals its successor, with the array based algorithm a thread t that completes its CAS scans the thread-records array starting from t's entry for finding a waiting thread to be signaled. This implies that every waiting thread will eventually receive the opportunity to attempt its read/CAS operations-pair.

Since the array-based algorithm does not use a dynamic waiting queue, threads may enter waiting mode and be signaled without having to perform a successful CAS on any of the *ArrayBasedCAS* data-structures. This is in contrast to MCS-CAS, where a thread must apply a successful CAS to the *tail* variable for joining the waiting queue.

Similarly to MCS-CAS, a thread t waits to be signaled for at most a platform-dependent period of time. If this waiting time expires, t proceeds with its read operation without further waiting. This ensures that the array-based algorithm is nonblocking. The full pseudo-code of the array-based algorithm and its description appears in Appendix B.

3. EVALUATION

We conducted our performance evaluation on SPARC and on Intel's Xeon and i7 multi-core CPUs. The SPARC machine comprises an UltraSPARC T2+ (Niagara II) chip containing 8 cores, each core multiplexing 8 hardware threads, for a total of 64 hardware threads. It runs the 64-bit Solaris 10 operating system with Java SE 1.6.0 update 23. The Xeon machine comprises a Xeon E7-4870 chip, containing 10 cores and hyper-threaded to 20 hardware threads. The i7 machine comprises an i7-920 CPU containing 4 cores and hyper-threaded to 8 hardware threads. Both Intel machines run the 64-bit Linux 3.2.1 kernel with Java SE 1.6.0 update 25. All tests were conducted with HotSpot in 64-bit *server* mode.

Initially we evaluated our CAS contention management algorithms using a synthetic CAS micro-benchmark and used the results to optimize the platform-dependent parameters used by the algorithms. We then evaluated the impact of our algorithms on implementations of widely-used data structures such as queues and stacks. No explicit threads placement was used.

3.1. The CAS micro-benchmark

To tune and compare our CAS contention management algorithms, we used the following synthetic *CAS benchmark*. For every concurrency level k, varying from 1 to the maximum number of supported hardware threads, k threads repeatedly read the same atomic reference and attempt to CAS its value, for a period of 5 seconds. Before the test begins, each thread generates an array of 128 random objects and during the test it attempts to CAS the value of the shared object to a reference to one of these objects, in a round-robin manner. In the course of the test, each thread

	Xeon	i7	Sparc
CB-CAS	WAITING_TIME=0.13ms	WAITING_TIME=0.8ms	WAITING_TIME=0.2ms

EXP-CAS	EXP_THRESHOLD=2
	c = 8 ; m = 24
		EXP_THRESHOLD=2
		c = 9 ; m = 27
			EXP_THRESHOLD=1
			c = 1 ; m = 15

MCS-CAS	CONTENTION_THRESHOLD=8
	NUM_OPS = 10,000
	MAX_WAIT = 0.9ms
		CONTENTION_THRESHOLD=8
		NUM_OPS = 10,000
		MAX_WAIT = 7.5ms
			CONTENTION_THRESHOLD=14
			NUM_OPS = 10
			MAX_WAIT = 1ms

AB-CAS	CONTENTION_THRESHOLD=2
	NUM_OPS = 10,000
	MAX_WAIT = 0.9ms
		CONTENTION_THRESHOLD=2
		NUM_OPS = 100,000
		MAX_WAIT = 7.5ms
			CONTENTION_THRESHOLD=14
			NUM_OPS = 100
			MAX_WAIT = 1ms

TS-CAS	CONC = 1 ; SLICE = 20	CONC = 1 ; SLICE = 25	CONC = 10 ; SLICE = 6

Table I. Summary of tuned algorithm parameters.

counts the number of successful CAS operations and these local counters are summed up at the end of the test.

Using the CAS benchmark, we've tuned the parameters used by the algorithms described in Section 2. The values that were chosen as optimal were those that produced the highest average throughput across all concurrency levels. These values appear in Table I.[5] Figures 2a-3b show the results of the CAS synthetic benchmarks on the three platforms on which we conducted our tests using these optimal parameter values. Each data point is the average of 10 independent executions.

3.1.1. Xeon results. Figure 2a shows the throughput (the number of successful CAS operations) on the Xeon machine as a function of the concurrency level. It can be seen that the throughput of Java CAS falls steeply for concurrency levels of 2 or more. Whereas a single thread performs approximately 413M successful CAS operations in the course of the test, the number of successful CAS operations is only approximately 89M for 2 threads and 62M for 4 threads. For higher concurrency levels, the number of successes remains in the range of 50M-59M operations.

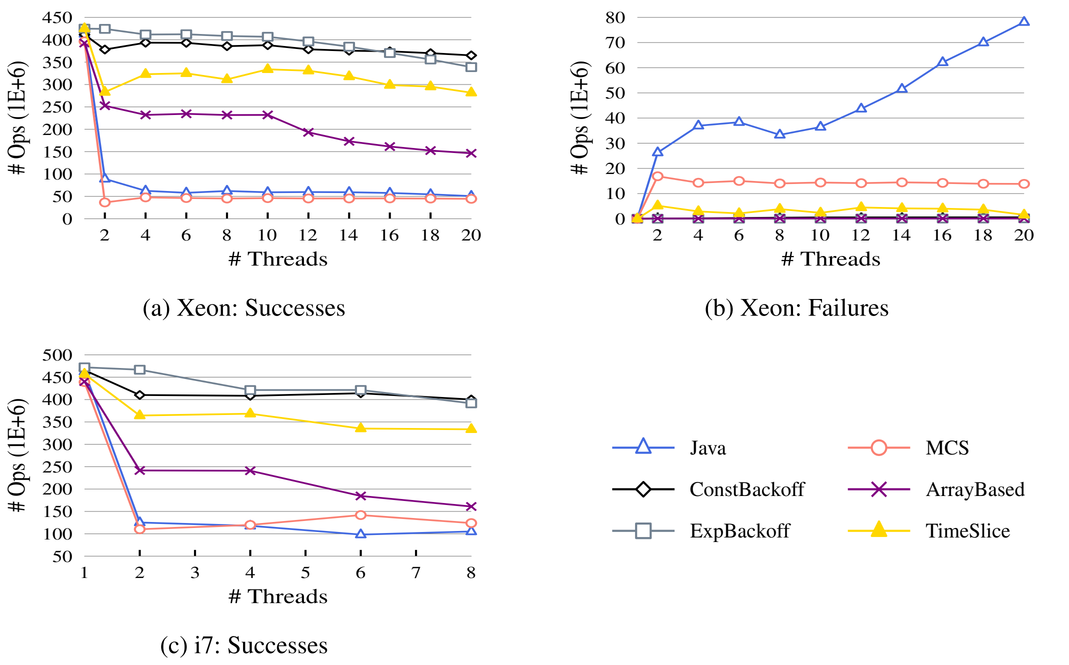
In sharp contrast, both the constant wait and exponential backoff CAS algorithms are able to maintain high throughput across the concurrency range. Exponential backoff is slightly better up until 16 threads, but then its throughput declines to less than 350M and falls below constant backoff. The throughput of both these algorithms exceeds that of Java CAS by a factor of more than 4 for 2 threads and their performance boost grows to a factor of between 6-7 for higher concurrency levels.

The time slice algorithm is the third performer in this test, outperforming Java CAS by a factor of between 3-5.6 but providing only between 65%-87% the throughput of constant and exponential backoff.

The array based algorithm incurs some overhead and performs only approximately 390M successful operations in the single thread tests. In higher concurrency levels, its throughput exceeds that of Java CAS by a factor of between 2.5-3 but it is consistently outperformed by the simpler backoff algorithms by a wide margin. MCS-CAS is the worst performer on the Xeon CAS benchmark and is outperformed by all other algorithms across the concurrency range.

More insights into these results are provided by Figure 2b, which shows the numbers of CAS failures incurred by the algorithms. All algorithms except for MCS-CAS incur orders-of-magnitude less failures than Java CAS. Specifically, for concurrency level 20, Java CAS incurs almost 80M CAS failures, constant backoff incurs approximately 569K failures and exponential backoff incurs approximately 184K failures. Array based incurs approximately 104K failures. MCS-CAS incurs a high number of failures since the tuning of its parameters sets the contention threshold to 8, implying that it is much less likely to enter high contention mode than array based. This high threshold indicates that MCS-CAS is not a good CAS contention management algorithm for Xeon.

[5]The values of the *WAITING_TIME* and *MAX_WAIT* parameters are expressed in milliseconds. Waiting is done by performing a corresponding number of loop iterations.



(a) Xeon: Successes

(b) Xeon: Failures

(c) i7: Successes

Figure 2. Xeon & i7 CAS: Number of successful and failed CAS ops as a function of concurrency level.

3.1.2. i7 results. Figure 2c shows the CAS throughput on the i7 machine as a function of the concurrency level. It can be seen that both the absolute and relative performance of the evaluated algorithms are very similar to the behavior on the Xeon machine. The numbers of CAS failures are also very similar to Xeon (for corresponding concurrency levels) and therefore a figure showing these numbers is not provided.

3.1.3. SPARC results. Figure 3a shows the throughput of the evaluated algorithms in the CAS benchmark on the SPARC machine. Unlike Xeon where Java CAS does not scale at all, on SPARC the performance of Java CAS scales from 1 to 4 threads but then quickly deteriorates, eventually falling to about 16% of the single thread performance, less than 9% of the performance of 4 threads. More specifically, in the single thread test, Java CAS performs slightly more than 48M successful CAS operations and its performance reaches a peak of almost 90M operations at 4 threads. Java CAS is the worst performer for concurrency levels 12 or higher and its throughput drops to approximately 8M for 64 threads.

The exponential backoff CAS is the clear winner on the SPARC CAS benchmark. Its throughput is slightly lower than that of Java CAS for concurrency levels 1 and 2, but for higher concurrency levels it outperforms Java CAS by a wide margin that grows with concurrency. For concurrency levels 28 or more, exponential backoff completes more than 7 times successful CAS operations and the gap peaks for 54 thread where Java CAS is outperformed by a factor of almost 12.

The constant wait CAS is second best. Since it has smaller overhead than exponential backoff CAS, it slightly outperforms it in the single thread test, but for higher concurrency levels it is outperformed by exponential backoff by a margin of up to 56%.

The high overhead of MCS-CAS and array-based CAS manifests itself in the single thread test, where both provide significantly less throughput than all other algorithms. For higher concurrency levels, both MCS-CAS and array based perform between 20M-60M successful CAS operations, significantly more than Java CAS but much less than the constant and exponential backoff algorithms.

Figure 3b shows the number of CAS failures incurred by the algorithms. Constant backoff and exponential bakcoff incur the smallest numbers of failures, an order of magnitude less failures

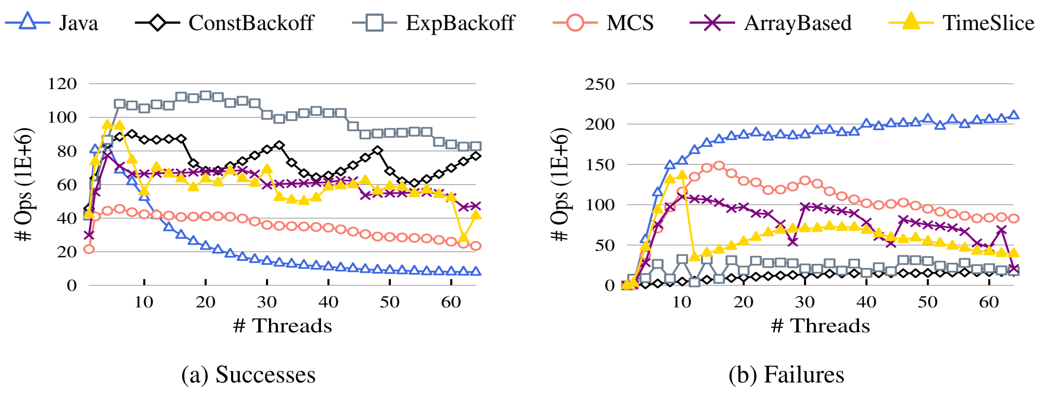


Figure 3. SPARC CAS: Number of successful and failed CAS ops as a function of concurrency level.

than Java CAS. Array based, time slice and MCS-CAS incur more failures than the two backoff algorithms, but significantly less than Java CAS in almost all concurrency levels.

Zooming into the numbers of successes and failures incurred by MCS-CAS in low- and high-contention modes, we find that for high concurrency levels, MCS-CAS obtains approximately 10% of its successes in high-contention mode but also incurs about 10 times more failures in low-contention mode than in high-contention mode.

3.1.4. Analysis. As shown by Figures 2a, 2c and 3a, whereas on SPARC the number of successes in the CAS benchmark scales up to 4 or 8 threads (depending on the contention management algorithm being used), no such scalability occurs on the Xeon or the i7 platforms. We now explain the architectural reasons for this difference. The explanation requires some background which we now provide.

The SPARC T2+ processor chip contains 8 cores where each core has a private 8KB L1 data cache and 2 pipelines with 4 hardware thread contexts per pipeline, for a total of 64 hardware thread contexts per chip. The L1 data caches, which are physically indexed and physically tagged, use a write-through policy where stores do not allocate. The 8 cores are connected via an intra-chip cross-bar to 8 L2 banks. Based on a hash of the physical address, the cross-bar directs requests to one of the 8 L2 cache banks. The L2 banks are 16-way set associative and have a total capacity of 4MB. Pairs of banks share DRAM channels. All store instructions, including CAS, pass over the cross-bar to the L2 cache. For coherence, the L2, which is inclusive of all L1s, maintains a reverse directory of which L1 instances hold a given line. L1 lines are either valid or invalid; there are no cache-to-cache transfers between L1 caches. T2+ processors enjoy very short cache-coherent communication latencies relative to other processors. On an otherwise unloaded system, a coherence miss can be satisfied from the L2 in less than 20 cycles.

CAS instructions are implemented at the interface between the cores and the cross-bar. For ease of implementation, CAS instructions, whether successful or not, invalidate the line from the issuer's L1. A subsequent load from that same address will miss in the L1 and revert to the L2. The cross-bar and L2 have sufficient bandwidth and latency, relative to the speed of the cores, to allow load-CAS benchmarks to scale beyond just one thread, as we see in Figure 3a.

We now describe why such scalability is not observed on the Xeon and i7 platforms, as seen by Figures 2a and 2c. Modern x86 processors tend to have deeper cache hierarchies, often adding core-local MESI L2 caches connected via an on-chip coherent interconnect fabric and backed by a chip-level L3. Intra-chip inter-core communication is accomplished by L2 cache-to-cache transfers. With respect to coherence, a store instruction is no different than a CAS – both need to issue request-to-own bus operations, if necessary, to make sure the underlying line can be modified. That is, CAS is performed "locally" in the L1 or L2.

In addition to the cost of obtaining ownership, load-CAS benchmarks may also be subject to a number of confounding factors on x86. As contention increases and the CAS starts to fail more frequently, branch predictors can be trained to expect the failure path, so when the CAS is ultimately successful the thread will incur a branch misprediction penalty. In contrast, T2+ does not have a branch predictor.

Furthermore, some x86 processors have an optimization that allows speculative coherence probes. If a load is followed in close succession, in program order, by a store or CAS to the same address, the processor may need to send coherence request messages to upgrade the line to writable state in its local cache at the time of the load. This avoids the situation where the load induces a read-to-share bus transaction followed in short order by a transaction to upgrade the line to writable state. While useful, under intense communication traffic this facility can cause excessive invalidation. Finally, we note that coherence arbitration for lines is not necessarily fair over the short term, and in turn this can impact performance.

3.1.5. Fairness. Table II summarizes the fairness measures of the synthetic CAS benchmarks. We used normalized standard deviation and Jain's fairness index [33] to quantify the fairness of individual threads' throughput for each concurrency level, and then took the average over all concurrency levels. The widely used Jain's index for a set of n samples is the quotient of the square of the sum and the product of the sum of squares by n. Its value ranges between $1/n$ (lowest fairness) and 1 (highest fairness). It equals k/n when k threads have the same throughput, and the other $n-k$ threads are starved. On Xeon, CB-CAS, and AB-CAS provide better fairness than Java CAS. On SPARC, CB-CAS and TS-CAS provide fairness comparable to that of Java. The average variability of the algorithms' throughput is quite low and does not exceed a single standard deviation.

	Normal stdev		Jain's Index	
	Xeon	SPARC	Xeon	SPARC
Java	0.291	0.164	0.900	0.961
CB-CAS	0.077	0.196	0.992	0.957
EXP-CAS	0.536	0.936	0.761	0.588
MCS-CAS	0.975	0.596	0.563	0.727
AB-CAS	0.001	0.822	1.000	0.638
TS-CAS	0.829	0.211	0.605	0.946

Table II. Fairness measures.

3.2. FIFO queue

To further investigate the impact of our CAS contention management algorithms, we experimented with the FIFO queue algorithm of Michael and Scott [11] (MS-queue). We used the Java code provided in Herlihy and Shavit's book [34] without any optimizations. The queue is represented by a list of nodes and by *head* and *tail* atomic references to the first and last entries in the list, which become hot spots under high contention.

We evaluated six versions of the MS-queue: one using Java's AtomicReference objects (called J-MSQ), and the other five replacing them by *ConstantBackoffCAS*, *ExpBackoffCAS*, *TimeSliceCAS*, *MSC-CAS* and *AB-CAS* objects (respectively called *CB-MSQ*, *Exp-MSQ*, *TS-MSQ*, *MSC-MSQ* and *AB-MSQ*), using the parameter values specified in Table I. We compared these algorithms with the Java 6 *ConcurrentLinkedQueue* class from the *java.util.concurrent* package,[6] and the flat-combining queue algorithm [7].[7] The *ConcurrentLinkedQueue* class implements an algorithm (henceforth called Java 6 queue) that is also based on Michael and Scott's algorithm. However, the Java 6 queue algorithm incorporates several significant optimizations such as performing lagged updates of the *head* and *tail* references and using *lazy sets* instead of normal writes.

We conducted the following test. For varying numbers of threads, each thread repeatedly performed either an *enqueue* or a *dequeue* operation on the data structure for a period of 5 seconds.

[6]We used a slightly modified version in which direct usage of Java's *Unsafe* class was replaced by an AtomicReference mediator.

[7]We used the Java implementation provided by Tel-Aviv University's Multicore Computing Group.

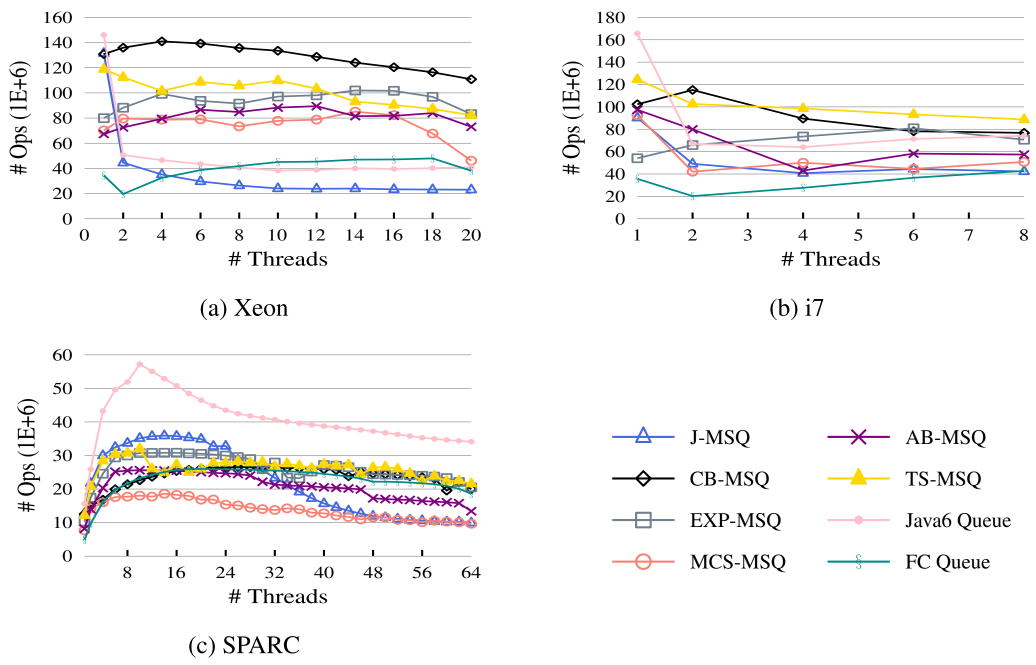


Figure 4. Queue: Number of completed ops as a function concurrency level.

The queue is pre-populated by 1000 items. A pseudo-random sequence of 128 integers is generated by each thread independently before the test starts, where the i'th operation of thread t is an *enqueue* operation if integer $(i \mod 128)$ is even and is a *dequeue* operation otherwise. Each thread counts the number of operations it completes on the queue. These local counters are summed up at the end of the test. Each data point is the average of 10 independent runs. In order to make the results comparable between the different platforms, the same set of 10 pre-generated seeds was used to initialize the random generator. Figures 4a-4c show the results of the queue tests on the three platforms we used for our experiments.

3.2.1. Xeon results. As shown by Figure 4a, CB-MSQ is the best queue implementation, outperforming the Java-CAS based queue in all concurrency levels by a factor of up to 6 (for 16 threads).

Surprisingly, CB-MSQ also outperforms the Java 6 queue by a wide margin in all concurrency levels except for 1, in spite of the optimizations incorporated to the latter. More specifically, in the single thread test, the performance of the Java 6 queue exceeds that of CB-MSQ by approximately 15%. In higher concurrency levels, however, CB-MSQ outperforms Java 6 queue by a factor of up to 3.5. Java 6 queue is outperformed in all concurrency levels higher than 1 also by EXP-MSQ and TS-MSQ. The FC queue hardly scales on this test and is outperformed by almost all algorithms in most concurrency levels. However, whereas in the Xeon CAS benchmark the constant backoff and exponential backoff provided nearly the same throughput, in the queue test CB-MSQ outperforms EXP-MSQ by a wide margin in most concurrency levels. J-MSQ has the worst performance in all concurrency levels higher than 1. It is outperformed by CB-MSQ by a factor of between 2-6 in all concurrency levels higher than 1. MSC-MSQ and AB-MSQ's performance is significantly inferior in comparison with CB-MSQ, EXP-MSQ and TS-MSQ but they outperform the other algorithms.

3.2.2. i7 results. Figure 4b shows the results of the queue test on the i7 machine. The differences between the algorithms in this test are less significant than on the Xeon. CB-MSQ and TS-MSQ provide the highest throughput for all concurrency levels except for 1. CB-MSQ peaks at 2 threads providing 124.5M operations, after which it starts to decline until reaching 81M for 8 threads. TS-MSQ maintains a consistent throughput of 90M-100M for concurrency levels higher than 1 and is the best performer for concurrency levels 6 or more. EXP-MSQ, which was significantly better than the Java 6 queue on Xeon, outperforms it only by roughly 5% in this test for concurrency levels of 2-6, and by 9% for 8 threads.

J-MSQ falls from 96.7M for 1 thread to about 40M-44M for higher concurrency levels, resembling its behavior on the Xeon. FC queue hardly scales in this test as well, providing the lowest throughput in all concurrency levels. TS-MSQ outperforms J-MSQ by a factor of between 2.1-2.4 for all concurrency levels except for 1.

3.2.3. SPARC results. Figure 4c shows the results of the queue test on the SPARC machine. Unlike on Xeon and i7, the Java 6 queue has the best throughput in all concurrency levels, outperforming TS-MSQ - which is second best in most concurrency levels - by a factor of up to 2. It seems that the optimizations of the Java 6 algorithm are more effective on the SPARC architecture. CB-MSQ starts low but its performance scales up to 30 threads where it slightly exceeds that of EXP-MSQ.

J-MSQ scales up to 14 threads where it performs approximately 36M queue operations, but quickly deteriorates in higher concurrency levels and its throughput falls below 10M operations at 64 threads. This is similar to the decline exhibited by Java CAS in the CAS benchmark, except that the graph is "stretched" and the decline is slightly milder. The reason for this change is that the effective levels of CAS contention on the data-structure's variables are reduced in the queue implementations, since the code of the MS-queue algorithm contains operations other than CAS. For concurrency levels 40 or higher, J-MSQ is outperformed by EXP-MSQ by a factor of up to 2.4 (for 54 threads). Unlike on Xeon, the FC queue scales on SPARC up to 20 threads, where its performance almost equals that of the simple backoff schemes. MCS-MSQ is the worst performer on this test and AB-CAS is only slightly better.

3.3. Stack

We also experimented with the lock-free stack algorithm of Treiber.[8] The stack is represented by a list of nodes and a reference to the top-most node is stored by an AtomicReference object. We evaluated six versions of the Treiber algorithm: one using Java's AtomicReference objects (called *J-Treiber*), and the other five replacing them by the *ConstantBackoffCAS*, *ExpBackoffCAS*, *TimeSliceCAS*, *MSC-CAS* and *AB-CAS* (respectively called *CB-Treiber*, *Exp-Treiber*, *TS-Treiber*, *MCS-Treiber* and *AB-Treiber*), using the parameter values specified in Table I. We also compared with a Java implementation of the elimination-backoff stack (*EB stack*) [13] (see Section 1.1).

The structure of the stack test is identical to that of the queue test: each thread repeatedly performs either a *push* or a *pop* operation on the stack for a period of 5 seconds. The stack is pre-populated by 1000 items. A pseudo-random sequence of 128 integers is generated by each thread independently before the test starts where the i'th operation of thread t is a *push* operation if integer $(i \mod 128)$ is even and is a *pop* operation otherwise. Each data point is the average of 10 independent runs. Figures 5a-5c show the results of the stack tests on the three platforms.

3.3.1. Xeon results. Figure 5a shows the results of the stack test on Xeon. As with all Xeon test results, also in the stack test, the implementation using Java's AtomicReference suffers from a steep performance decrease as concurrency levels increase, falling from throughput of approximately 126M stack operations in the single thread test to approximately 17M operations for 20 threads, approximately 13% of the single thread performance.

The EB stack is the winner of the Xeon stack test and CB-Treiber is second-best lagging behind only slightly. CB-Treiber maintains and even exceeds its high single-thread throughput across the concurrency range, scaling up from 144M operations for a single thread to 195M operations for 18 threads, outperforming J-Treiber by a factor of 11.5 for 18 threads. The performance of the EXP-Treiber and TS-Treiber algorithms lags behind that of CB-Treiber in all concurrency

[8]The first non-blocking implementation of a concurrent list-based stack appeared in [35] and used the double-width compare-and-swap (CAS) primitive. Treiber's algorithm is a variant of that implementation, in which *push* operations use a single-word-width CAS.

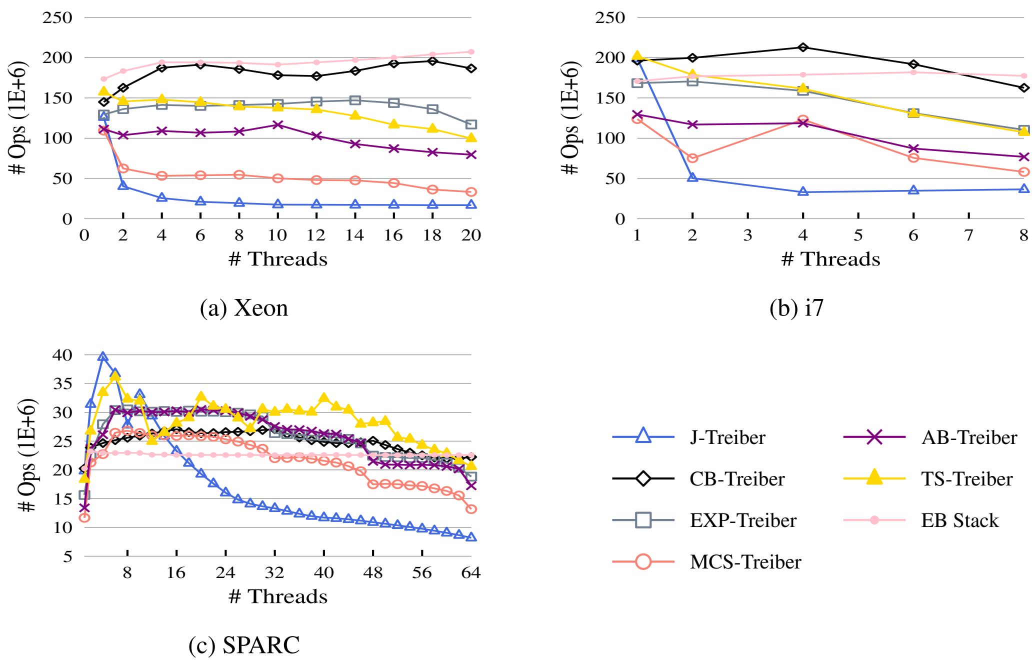


Figure 5. Stack: Number of completed ops as a function concurrency level.

levels by between 20%-40%. MCS-Treiber and AB-Treiber perform better than J-Treiber but are outperformed by all other algorithms.

3.3.2. i7 results. Figure 5b shows the results of our evaluation on the i7. The EB stack and CB-Treiber algorithms are the best performers. CB-Treiber has the upper hand in low concurrency levels, providing between 5%-10% more throughput than EB stack for 1-4 threads. It scales up to 4 threads, then starts to deteriorate and levels up with EB stack at 6 threads, where the throughput of both algorithms is approximately 186M. EB Stack maintains a consistent throughput of approximately 185M operations throughout the whole concurrency range, outperforming CB-Treiber at 8 threads by 15%.

EXP-Treiber is significantly outperformed by both EB stack and CB-Treiber. Its throughput declines from approximately 168M in the single-thread test to approximately 110M for 8 threads. TS-Treiber starts high at 206M for 1 thread, but deteriorates to 107M, in high correlation with Exp-Treiber. J-Treiber exhibits a similar behavior to the corresponding Java CAS in the CAS benchmark; it falls from more than 212M for a single thread to only 37M for 8 threads, and is outperformed by CB-Treiber in all concurrency levels except for 1 by a wide margin of up to 6.2. The relative performance of MCS-Treiber and AB-Treiber is very similar to that on Xeon.

3.3.3. SPARC results. Figure 5c shows the results of the stack tests on SPARC. J-Treiber scales up to 6 threads where it reaches its peak performance of 39.5M stack operations. Then its performance deteriorates with concurrency and reaches less than 10M operations for 64 threads. From concurrency level 18 and higher, J-Treiber has the lowest throughput. TS-Treiber has the highest throughput in most medium and high concurrency levels, with EXP-Treiber mostly second best. Unlike on Xeon, EB stack is almost consistently and significantly outperformed on SPARC by all simple backoff algorithms.

TS-Treiber has the highest throughput for 30 threads or more (with the exception of concurrency levels 62-64) and outperforms J-Treiber in high concurrency levels by a factor of up to 3. CB-Treiber starts low but scales up to 18 threads where it stabilizes at approximately 27M operations until it begins to decline at 34 threads and higher, matching and even slightly exceeding TS-Treiber and EXP-Treiber at 62 and 64 threads. MCS-Treiber is outperformed by all algorithms except for J-Treiber. The performance of AB-Treiber in this test is similar to that of CB-Treiber.

4. DISCUSSION

We conduct what is, to the best of our knowledge, the first study on the impact of contention management algorithms on the efficiency of the CAS operation. We implemented several Java classes that encapsulate calls to Java's AtomicReference class by CAS contention management algorithms. We then evaluated the benefits gained by these algorithms on the Xeon, SPARC and i7 platforms by using both a synthetic benchmark and CAS-based concurrent data-structure implementations of stacks and queues.

The three simplest algorithms that we've implemented - constant backoff, exponential backoff and time-slice - yielded the best results, primarily because they have very small overheads. The more complicated approaches - the MCS-CAS and array-based algorithms - provided better results than direct calls to AtomicReference in most cases, but were significantly outperformed by the simpler algorithms. As the performance margin is maintained and even increased for high concurrency levels, it is reasonable to assume that the simpler algorithms will remain superior also for larger numbers of threads.

We also compared relatively simple data-structure implementations that use our CAS contention management classes with more complex implementations that employ data-structure specific optimizations and use AtomicReference objects. We have found that, in some cases, simpler and non-optimized data-structure implementations that apply efficient contention management for CAS operations yield better performance than that of highly optimized implementations of the same data-structure that use Java's AtomicReference directly.

Our results imply that encapsulating invocations of CAS by lightweight contention management classes is a simple and generic way of significantly improving the performance of CAS-based concurrent objects. Adding these classes to the Java libraries will allow programmers to evaluate the extent to which their code may benefit from these algorithms.

This work can be extended in several directions. First, we may have overlooked CAS contention management algorithms that may yield even better results. Second, our methodology used per-architecture manual tuning of the contention management algorithm parameters by using the results of the CAS micro-benchmark. Although the generality of this approach is appealing, tuning these parameters per data-structure may yield better results. Moreover, dynamic tuning may provide a more general, cross data-structure, cross platform solution.

It would also be interesting to investigate if and how similar approaches can be used for other atomic-operation related classes in both Java and other programming languages such as C++. Finally, combining contention management algorithms at the atomic operation level with optimizations at the data-structure algorithmic level may yield more performance gains than applying each of these approaches separately. We leave these research directions for future work.

ACKNOWLEDGEMENTS

We thank Yehuda Afek, Nir Shavit and Tel-Aviv University's Multicore Computing Group for kindly allowing us to use their servers for our evaluation.

REFERENCES

1. Attiya H, Welch J. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* John Wiley Interscience, 2004.
2. Herlihy M, Lim BH, Shavit N. Scalable concurrent counting. *ACM Trans. Comput. Syst.* 1995; **13**(4):343–364.
3. Herlihy M, Shavit N, Waarts O. Linearizable counting networks. *Distributed Computing* 1996; **9**(4):193–203.
4. Afek Y, Hakimi M, Morrison A. Fast and scalable rendezvousing. *DISC*, 2011; 16–31.
5. Fatourou P, Kallimanis ND. Revisiting the combining synchronization technique. *PPOPP*, 2012; 257–266.

6. Gidenstam A, Sundell H, Tsigas P. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. *OPODIS*, 2010; 302–317.
7. Hendler D, Incze I, Shavit N, Tzafrir M. Flat combining and the synchronization-parallelism tradeoff. *SPAA*, 2010; 355–364.
8. Hendler D, Incze I, Shavit N, Tzafrir M. Scalable flat-combining based synchronous queues. *DISC*, 2010; 79–93.
9. III WNS, Lea D, Scott ML. Scalable synchronous queues. *Commun. ACM* 2009; **52**(5):100–111.
10. Ladan-Mozes E, Shavit N. An optimistic approach to lock-free fifo queues. *Distributed Computing* 2008; **20**(5):323–341.
11. Michael MM, Scott ML. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *PODC*, 1996; 267–275.
12. Moir M, Nussbaum D, Shalev O, Shavit N. Using elimination to implement scalable and lock-free fifo queues. *SPAA*, 2005; 253–262.
13. Hendler D, Shavit N, Yerushalmi L. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.* 2010; **70**(1):1–12.
14. Afek Y, Korland G, Natanzon M, Shavit N. Scalable producer-consumer pools based on elimination-diffraction trees. *Euro-Par (2)*, 2010; 151–162.
15. Basin D, Fan R, Keidar I, Kiselov O, Perelman D. Café: Scalable task pools with adjustable fairness and contention. *DISC*, 2011; 475–488.
16. Gidron E, Keidar I, Perelman D, Perez Y. Salsa: scalable and low synchronization numa-aware algorithm for producer-consumer pools. *SPAA*, 2012; 151–160.
17. Goodman EL, Lemaster MN, Jimenez E. Scalable hashing for shared memory supercomputers. *SC*, 2011; 41.
18. Herlihy M, Shavit N, Tzafrir M. Hopscotch hashing. *DISC*, 2008; 350–364.
19. Shalev O, Shavit N. Split-ordered lists: Lock-free extensible hash tables. *J. ACM* 2006; **53**(3):379–405.
20. Triplett J, McKenney PE, Walpole J. Scalable concurrent hash tables via relativistic programming. *Operating Systems Review* 2010; **44**(3):102–109.
21. Herlihy M. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems* Jan 1991; **13**(1):123–149.
22. Corporation I. *Intel Itanium Architecture Software Developer's Manual*. 2006.
23. Microsystems S. *UltraSPARC Architecture 2005, Draft D0.9.2*. 2008.
24. Motorola. *MC68000 Programmer's Reference Manual*. 1992.
25. Dice D, Lev Y, Moir M. Scalable statistics counters. *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, ACM: New York, NY, USA, 2013; 43–52, doi:10.1145/2486159.2486182.
26. Dice D, Hendler D, Mirsky I. Lightweight contention management for efficient compare-and-swap operations. *Euro-Par*, 2013; 595–606.
27. *Bitbucket repository*. URL `https://bitbucket.org/mirsky/scalablecas`.
28. Herlihy M, Luchangco V, Moir M, Scherer WN III. Software transactional memory for dynamic-sized data structures. *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, ACM: New York, NY, USA, 2003; 92–101.
29. Guerraoui R, Herlihy M, Pochon B. Toward a theory of transactional contention managers. *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, ACM: New York, NY, USA, 2005; 258–264, doi:10.1145/1073814.1073863.
30. Anderson TE. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* Jan 1990; **1**(1):6–16, doi:10.1109/71.80120. URL `http://dx.doi.org/10.1109/71.80120`.
31. Mellor-Crummey JM, Scott ML. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 1991; **9**(1):21–65, doi:http://doi.acm.org/10.1145/103727.103729.
32. Craig T. Building fifo and priority-queuing spin locks from atomic swap. *Technical Report* 1993.
33. Vandalore B, Fahmy S, Jain R, Goyal R, Goyal M. A definition of general weighted fairness and its support in explicit rate switch algorithms. *ICNP*, 1998; 22–30.
34. Herlihy M, Shavit N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2008.
35. IBM. *IBM System/370 Extended Architecture, Principles of Operation, publication no. SA22-7085*. 1983.

APPENDIX A: THE MCS-CAS ALGORITHM

Algorithm 4 presents the *MCS-CAS* class, which implements the MCS-CAS algorithm. Each class instance contains the following two fields. The *tail* field is an atomic integer storing the *TInd* of the thread that is at the tail of the queue of threads that are currently in high-contention mode. The *tRecords* field is an array of per-thread data records, storing the following fields.[9] The *contentionMode* field is a boolean, indicating whether the respective thread is in high-contention mode (if true) or in low-contention mode (if false). The *next* field of the record corresponding to t is an atomic integer, used by t's successor in the queue for communicating its *TInd* to t. The *notify* field of the record corresponding to t is an atomic integer array used by t's predecessor to signal t when it is allowed to proceed with its read operation. The *modeCount* field is used by a thread to determine when it should shift from high-contention mode to low-contention mode or vice versa as we

[9]To cope with false sharing, the records are padded with dummy fields.

soon explain. We start by describing the *read* operation. If thread t is in low-contention mode, then it simply delegates to the *get* method of the AtomicReference object to return the current reference value in line 64.

If t is in high-contention mode, then it initializes its *next* entry (line 54) and swaps the value of *tail* to its *TInd* (line 55). After the swap t checks if it has a predecessor (line 56). If it does, t writes its *TInd* to the *next* field of the *pred* entry of its predecessor in the *tRecords* array, and initializes its *notify* field (lines 57–58).

Thread t then waits until it is either notified by its predecessor that it can go ahead or until a platform-dependent waiting time elapses (lines 59–60) and then returns the current reference value (line 64).

We now describe the *CAS* operation. First, the *compareAndSet* method on the AtomicReference superclass is called, passing to it the *old* and *new* operands (line 67). If t is in high-contention mode then it checks whether a successor has written its *TInd* to t's *next* field (line 70) and if so signals that successor that it may stop waiting for t (lines 80–81).

If no successor wrote its *TInd*, then t attempts to swap the field *tail* back from its *TInd* to *NONE* in line 71. If it fails, then it has a successor, in which case t waits for it to write its *TInd* for at most a platform-dependent period of time (lines 72–73) and then re-checks its *next* field; if it's non-empty, t signals the successor (lines 75–77).

Before exiting the CAS method, t increments its *modeCount* field and shifts to low-contention mode if the number of CAS operations it applied in high-contention mode reached a platform-dependent threshold value (lines 84–87).

When t is in low-contention mode, its *modeCount* field is used for counting the number of consecutive CAS failures. If the current CAS operation was successful, then t resets *modeCount* field (line 89). If the CAS failed then the field is incremented. If the number of consecutive failures now reaches a platform-dependent threshold value, t shifts to high-contention mode (lines 90–94).

Regardless of whether t is in high- or low-contention mode it always leaves the function by returning the CAS success/failure indication in line 97.

APPENDIX B: THE ARRAY-BASED CAS ALGORITHM

Algorithm 5 presents the *ArrayBasedCAS* class, which implements the array-based CAS contention management algorithm. We now describe the algorithm in detail.

Similarly to the MCS-CAS algorithm, with the array based CAS threads may apply their operations in either low-contention or high-contention mode. Each class instance contains the following fields. The *tRecords* array stores for each thread the following fields; *contentionMode*, *request* and *modeCount*, which are used by the array based algorithm similarly to the way they are used by MCS-CAS. The *owner* atomic integer stores the *TInd* of the current "owner" of the memory location or *NONE* if there is no such owner. At any point in time, the owner thread is the single high-contention mode thread that is permitted to perform *read* or *CAS* operations on the memory location encapsulated by the *ArrayBasedCAS* object without waiting.

We now describe the *read* operation. If thread t is in low-contention mode or is the current owner (line 109), then it simply delegates to the *get* method of the AtomicReference object to return the current reference value (line 118). If t is in high-contention mode and is not the owner, then it initializes its *request* entry to true (line 131) and executes the loop at lines 111–115, until it is either signaled, manages to become the owner, or performs a platform-dependent number of loop iterations. If t is signaled or becomes the owner in the course of the loop then it immediately exits it, ensuring that its *request* entry is reset in line 113 or line 116. After exiting the loop, t returns the current reference value in line 118.

We now describe the *CAS* operation. First, the *compareAndSet* method on the AtomicReference superclass is called, passing to it the *old* and *new* operands (line 121). If t is in high-contention mode (line 123), then it is the current owner. An owner performs *NUM_OP* (a platform-dependent value) number of CAS operations before releasing ownership. Thread t increments its *modeCount* field (line 124). If it has to release ownership, then it resets its *modeCount* field and exits high-contention mode (lines 125–126). It then scans the *tRcords* array and notifies the next waiting thread (if any) that it now becomes the owner (lines 128–134). If no waiting thread is found, t sets the value of the *owner* field to *NONE* (line 135).

If t is not in high-contention mode, then it proceeds to update its statistics. If the current CAS was successful (line 137), then t resets its *modeCount* field line 138. If the current CAS failed (line 140), thread t increments its *modeCount* field which counts the number of consecutive failures in low-contention mode. If this number now reaches a platform-dependent threshold value (line 140), t resets its *modeCount* entry and shifts to high-contention mode (lines 141–142).

Regardless of whether t is in high- or low-contention mode it always leaves the method by returning the CAS success/failure indication in line 144.

Algorithm 4: The `MCS-CAS` class.

```
42 public class MCS-CAS<V> extends AtomicReference<V>
43 private class ThreadRecord {
44     long modeCount
45     boolean contentionMode
46     int next = NONE
47     volatile boolean notify
48 }
49 private ThreadRecord[] tRecords = new ThreadRecord[MAX_THREADS]
50 private AtmicInteger tail = new AtomicInteger(NONE)
51 public V read() {
52     ThreadRecord r = tRecords[TInd]
53     if r.contentionMode then
54         r.next = NONE
55         int pred = tail.getAndSet(TInd)
56         if pred != NONE then
57             tRecords[pred].next.set(TInd)
58             r.notify.set(false)
59             long wait = MAX_WAIT
60             while ¬r.notify[TInd] ∧ (wait > 0) do
                   wait = wait-1
61
62         end
63     end
64     return get()
65 }
66 public boolean CAS(V old, V new) {
67     boolean ret = compareAndSet(old,new)
68     ThreadRecord r = tRecords[TInd]
69     if r.contentionMode then
70         if r.next == NONE then
71             if ¬tail.compareAndSet(TInd, NONE)
               then
72                 long wait = MAX_WAIT
73                 while r.next == NONE ∧ (wait > 0)
                   do wait = wait-1
74
75                 int successor = r.next
76                 if successor ≠ NONE then
77                     tRecords[successor].notify =
                       true
78                 end
79             else
80                 int successor = r.next[TInd]
81                 tRecords[successor].notify = true
82             end
83         end
84         r.modeCount = r.modeCount + 1
85         if r.modeCount = NUM_OPS then
86             r.contentionMode = false
87             r.modeCount = 0
88         end
89     else if ret then r.modeCount = 0
90     else
91         r.modeCount = r.modeCount + 1
92         if r.modeCount ==
           CONTENTION_THRESHOLD then
93             r.contentionMode = true
94             r.modeCount = 0
95         end
96     end
97     return ret
98 }
```

Algorithm 5: The `ArrayBasedCAS` class.

```
99  public class AB-CAS<V> extends AtomicReference<V>
100 private class ThreadRecord {
101     long modeCount
102     boolean contentionMode
103     volatile boolean request
104 }
105 private ThreadRecord[] tRecords = new ThreadRecord[MAX_THREADS]
106 private AtomicInteger owner = new AtomicInteger(NONE)
107 public V read() {
108     ThreadRecord r = tRecords[TInd]
109     if r.contentionMode ∧ (owner.get() ≠ TInd) then
110         r.request = true
111         for i=0; (i<MAX_WAIT) ∧ r.request; i++ do
112             if owner.get() == NONE ∧
                owner.compareAndSet(NONE, TInd)
                then
113                 r.request = false , break
114             end
115         end
116         if r.request then r.request = false
117     end
118     return get()
119 }
120 boolean CAS(V old, V new) {
121     boolean ret = compareAndSet(old,new)
122     ThreadRecord r = tRecords[TInd]
123     if r.contentionMode then
124         if ++r.modeCount ≥ NUM_OPS then
125             r.modeCount
126             r.contentionMode = false
127             int MT = MAX_THREADS
128             for i = (TInd+1)%MT; i ≠ TInd; i =
                (i+1)%MT do
129                 if tRecords[i].request then
130                     owner.set(i)
131                     r.request = false
132                     return ret
133                 end
134             end
135             owner.set(NONE)
136         end
137     else if ret then
138         r.modeCount = 0
139     end
140     else if ++r.modeCount ≥
        CONTENTION_THRESHOLD then
141         r.modeCount = 0
142         r.contentionMode = true
143     end
144     return ret
145 }
```