Lightweight Contention Management for Efficient Compare-and-Swap Operations*

Dave Dice[1], Danny Hendler[2] and Ilya Mirsky[3]

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

Abstract. Many concurrent data-structure implementations 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 the degradation in its performance in the presence of memory contention.

In this work we study the following question: can software-based contention management improve the efficiency of hardware-provided CAS operations? Our performance evaluation establishes that lightweight contention management support can greatly improve performance under medium and high contention levels while typically incurring only small overhead when contention is low.

Keywords: 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* [3,13]: abstract data structures that are concurrently accessed by asynchronous threads. Efficient concurrent data structure algorithms 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., [11,14]), queues (e.g.,[1,8,18], and stacks (e.g.,[6,8]), 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 [15] - and is supported by hardware in most contemporary multiprocessor architectures. The CAS operation takes three arguments: a memory

* 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.

address[4], 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.

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. 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": as failed CAS operations generate coherence traffic on most architectures, they 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 plus (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.[5] 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 platform.

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, significantly improve the performance of CAS-based concurrent data-structures?

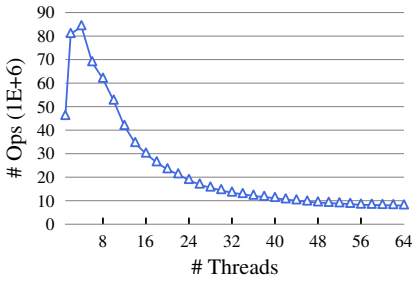


Fig. 1: SPARC: Java's CAS

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 native CAS by *contention management classes*. 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

[4] 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.

[5] We provide more details on this test in Section 3.

and i7 platforms by using both a synthetic micro-benchmark and CAS-based concurrent data-structure implementations of stacks and queues.[6]

The idea of employing contention management and backoff techniques to improve performance was widely studied in the context of software transactional memory (see, e.g., [12,7]) and lock implementations (see, e.g., [2,17,4]). Backoff techniques are also used at the higher abstraction level of specific data structures implementations [9,10,18]. However, this approach adds complexity to the design of the data-structure and requires careful per-data structure tuning. Our approach, of adding contention management (and, specifically, backoff) mechanisms at the CAS instruction level, provides a simple and generic solution, in which tuning can be done *per architecture* rather than per implementation.

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 up to a factor of 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 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.[7]

[6] 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.

[7] None of the methods of *AtomicReference* are overridden.

In some of our algorithms, threads need to access per-thread state associated with the object. For example, a thread may keep a 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.[8]

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.

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 2). The *CAS* operation invokes the *compareAndSet* method on the *AtomicReference* superclass, passing to it the *old* and *new* operands (line 4). The *CAS* operation returns *true* in line 7 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 5–6).

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 9). The *CAS* operation invokes the *compareAndSet* method on the *AtomicReference* superclass (line 11). If the CAS is successful, the method returns *true* (line 12). If the CAS fails and the number of registered threads exceeds a platform-dependent level *CONC* (line 13), 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 the range $\{1, \ldots, \lceil regN/CONC \rceil\}$ (line 14). 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 starts and then returns *false* (lines 14–17).

[8] 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 1: `ConstBackoffCAS`

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

Algorithm 2: `TimeSliceCAS`

```
8  public class TimeSliceCAS<V>
   extends AtomicReference<V>;
9  public V read()  { return get() }
10 public boolean CAS(V old, V new)
11     if compareAndSet(old,new) then
12         return true
13     if regN > CONC then
14         int sliceNum =
           Random.nextInt(⌈regN/CONC⌉)
           repeat
15             currentSlice =
               (System.nanoTime() >>
               SLICE) % ⌈regN/CONC⌉;
16         until sliceNum = currentSlice;
17     return false ;
```

Algorithm 3: `ExpBackoffCAS`

```
18 public class ExpBackoffCAS<V>
   extends AtomicReference<V>;
19 private int[] failures = new int
   [MAX_THREADS] ;
20 public V read()  { return get() }
21 public boolean CAS(V old, V new)
22     if compareAndSet(old,new) then
23         if failures[TInd] > 0 then
24             failures[TInd]−−;
25         return true
26     else
27         int f = failures[TInd]++ ;
28         if f > EXP_THRESHOLD then
           wait(2^min(c·f,m));
29         return false;
```

Fig. 2: Fairness measures

	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

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 19). The *read* operation simply delegates to the *get* method of the *AtomicReference* class (line 20).

The *CAS* operation invokes the *compareAndSet* method on the *AtomicReference* superclass (line 22). If the CAS is successful, then the *CAS* operation returns *true* (line 25).

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 27–28), eventually returning *false*.

The MCS-CAS and ArrayBasedCAS Algorithms : *MCS-CAS* and *ArrayBasedCAS* are described here briefly for lack of space. They are much more complex than the first 3 algorithms we described. Pseudo codes and full descriptions appear in a technical report [5]. *MCS-CAS* implements a variation of the Mellor-Crummey and Scott (MCS) lock algorithm [17] to serialize load-CAS operations. 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.

The *ArrayBasedCAS* algorithm uses an array-based signaling mechanism, in which a *lock owner* searches for the next entry in the array, on which a thread is waiting for permission to proceed with its load-CAS operations, in order to signal it. Also in this algorithm, waiting-times are bounded.

3 Evaluation

We conducted our performance evaluation on the SPARC, Intel's Xeon and i7 multi-core CPUs. The i7 results are very similar to those on Xeon. For lack of space, they are only described in the technical report [5]. The SPARC machine comprises an UltraSPARC T2 plus (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. It runs the 64-bit Linux 3.2.1 kernel with Java SE 1.6.0 update 25.

Initially we evaluated our CAS contention management algorithms using a synthetic 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.

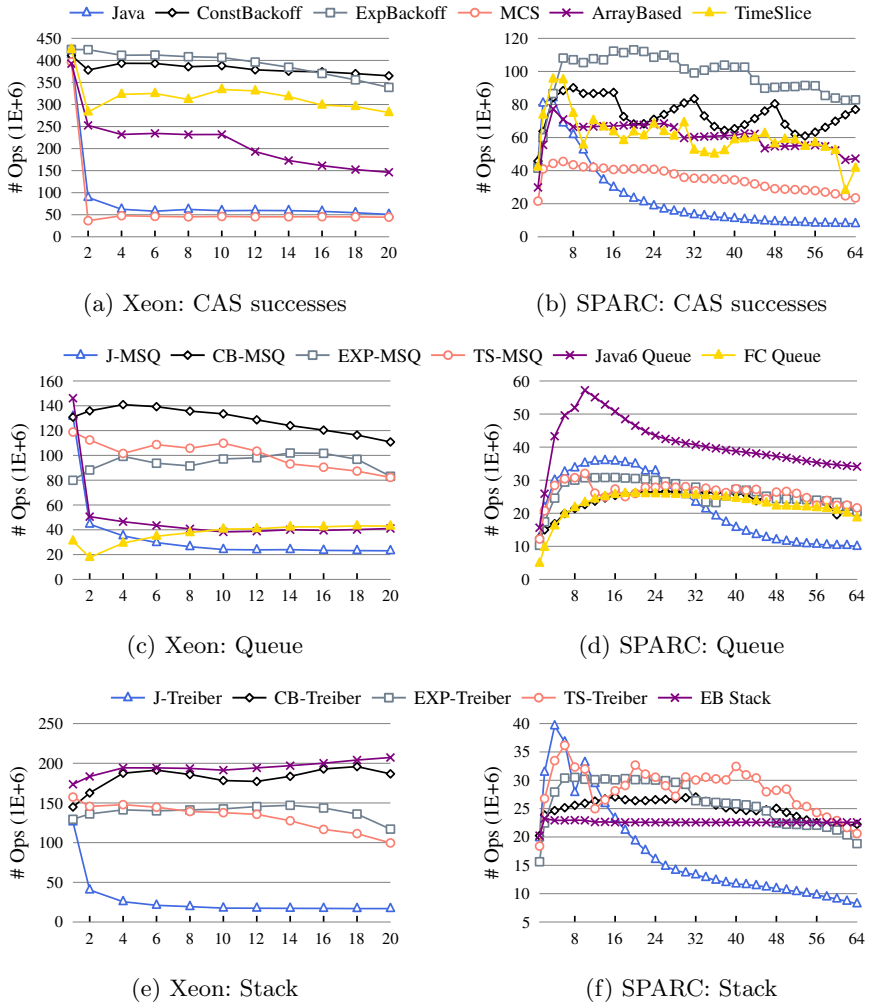
3.1 The CAS Micro-Benchmark

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 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.

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 of all concurrency levels. [9]

Figure 3a shows the throughput (the number of successful CAS operations) on the Xeon machine as a function of the concurrency level. Each data point is the average of 10 independent executions. 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-58M operations.

[9] A table with the values of tuned parameters is provided in [5].



(a) Xeon: CAS successes

(b) SPARC: CAS successes

(c) Xeon: Queue

(d) SPARC: Queue

(e) Xeon: Stack

(f) SPARC: Stack

Fig. 3: Benchmark results: throughput as a function of concurrency level.

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 below 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 3'rd 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 and MCS-CAS algorithms significantly lag behind the simpler back-

off algorithms. More insights into the results of these tests are provided in the technical report [5].

Figure 3b 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. Java CAS is the worst performer for concurrency levels 12 or higher and its throughput drops to about 8M for 64 threads. The exponential backoff CAS is the clear winner on the SPARC CAS benchmark. For concurrency levels 28 or more, exponential backoff completes more than 7 times successful CAS operations and the gap peaks for 54 threads where Java CAS is outperformed by a factor of almost 12. The constant wait CAS is second best. 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 CAS perform between 30M-60M successful CAS operations, significantly more than Java CAS but much less than the constant and exponential backoff algorithms.

Analysis : As shown by Figures 3a and 3b, whereas the number of successes in the CAS benchmark on the SPARC scales up to 4 or 8 threads (depending on the contention management algorithm being used), no such scalability occurs on the Xeon. In the technical report [5] we provide a detailed explanation of the architectural reasons for this difference. For lack of space, we only provide a brief explanation here.

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 cache in under 20 cycles. CAS instructions are implemented on SPARC 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 3b.

We now describe why such scalability is not observed on the Xeon platform, as seen by Figure 3a. 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. In addition to the cost of obtaining cache-line ownership, load-CAS benchmarks may also be subject to a number of additional confounding factors on x86 which we describe in the technical report [5].

Fairness : Table 2 summarizes the fairness measures of the synthetic CAS benchmarks. We used normalized standard deviation and Jain's fairness index [16] to assess 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. We see that CB-CAS and TS-CAS provide comparable and even superior fairness to Java CAS while the rest of the algorithms provide less fairness.

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 [18] (MS-queue).[10] 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 four versions of the MS-queue: one using Java's *AtomicReference* objects (called J-MSQ), and the other three replacing them by *ConstantBackoffCAS*, *ExpBackoffCAS* and *TimeSlice* objects (respectively called CB-MSQ, Exp-MSQ and TS-MSQ). We also compared with the flat-combining queue algorithm [8]. MCS-CAS and array based CAS were consistently outperformed and are therefore omitted from the following comparison. We compared these algorithms with the Java 6 *java.util.concurrent.ConcurrentLinkedQueue* class.[11] The *ConcurrentLinkedQueue* class implements an algorithm (henceforth simply 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 number of threads, each thread repeatedly performed either an *enqueue* or a *dequeue* operation on the data structure for a period of 5 seconds. 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.

Figures 3c and 3d show the results of the queue tests on the Xeon and SPARC platforms. As shown by Figure 3c, CB-MSQ is the best queue implementation on Xeon, outperforming the *AtomicReference*-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 1, in spite of the optimizations incorporated to the latter. More specifically, in the single thread test Java 6 queue performance exceeds that of CB-MSQ by approximately 15%. In higher concurrency levels, however, CB-MSQ outperforms Java 6 queue

[10] We used the Java code provided in Herlihy and Shavit's book [13] without any optimizations.

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

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.

Figure 3d shows the results of the queue tests on the SPARC machine. Here, unlike on Xeon, 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 22 threads where it almost matches that of EXP-MSQ.

J-MSQ scales up to 12 threads where it performs approximately 35M queue operations, but quickly deteriorates in higher concurrency levels. For concurrency levels 50 or higher, J-MSQ is outperformed by CB-MSQ by a factor of 2 or more. Unlike on Xeon, the FC queue scales on SPARC up to 24 threads, when its performance almost equals that of the simple backoff schemes.

3.3 Stack

We also experimented with the lock-free stack algorithm of Treiber [19]. The Treiber stack is represented by a list of nodes and a reference to the top-most node is stored by an *AtomicReferece* object. We evaluated the following versions of the Treiber algorithm: one using Java's *AtomicReference* objects (called J-Treiber), and the other three replacing them by the *ConstantBackoffCAS*, *ExpBackoffCAS* and *TimSli*ceCAS (respectively called CB-Treiber, Exp-Treiber and TS-Treiber). We also compared with a Java implementation of the elimination-backoff stack (EB stack) of Hendler et al. [9].[12] The structure of the Stack test is identical to that of the Queue test.

Figure 3e 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. 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 less than 150M operations for a single thread to almost 200M operations for higher concurrency levels, outperforming J-Treiber by a factor of more than 10 for high concurrency levels. TS-Treiber and EXP-Treiber are significantly outperformed by the EB stack and CB-Treiber algorithms.

Figure 3f shows the results of the stack tests on SPARC. J-Treiber scales up to 6 threads where it reaches its peak performance. Then its performance deteriorates with concurrency and reaches less than 10M operations for 64 threads. From concurrency level 16 and higher, J-Treiber has the lowest throughput. TS-Treiber has the highest throughput in most medium and high concurrency

[12] We used IBM's implementation available from the Amino Concurrent Building Blocks project at http://amino-cbbs.wiki.sourceforge.net/

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.

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.

Out of the contention management approaches we have experimented with, the three simplest algorithms - constant backoff, exponential backoff and timeslice - yielded the best results, primarily because they have very small overheads. The more complicated approaches - the MCS-CAS and array based CAS algorithms - provided better results than direct calls to *AtomicReferece* in most tests, but were significantly outperformed by the simpler approaches.

Our evaluation demonstrates that encapsulating Java's *AtomicReference* by objects that implement lightweight contention management support can improve the performance of CAS-based algorithms considerably.

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 improving the performance of concurrent objects.

This work may be extended in several directions. First, we may have overlooked CAS contention management algorithms that yield better results. Second, our methodology tuned the platform-dependent parameters of contention management algorithms by using the CAS benchmark. Although the generality of this approach is appealing, tuning these parameters per data-structure may yield better results. Moreover, a dynamic tuning may provide a general, cross data-structure, cross CPU, 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/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 only one of these approaches separately. We leave these research directions for future work.

Acknowledgements

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

References

1. Yehuda Afek, Michael Hakimi, and Adam Morrison. Fast and scalable rendezvousing. In *DISC*, pages 16–31, 2011.
2. T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.
3. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.
4. TravisS. Craig. Building fifo and priority-queuing spin locks from atomic swap. Technical report, 1993.
5. D. Dice, D. Hendler, and I. Mirsky. Lightweight Contention Management for Efficient Compare-and-Swap Operations. *ArXiv e-prints*, May 2013.
6. Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *PPOPP*, pages 257–266, 2012.
7. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC*, pages 258–264, New York, NY, USA, 2005. ACM.
8. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364, 2010.
9. Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.*, 70(1):1–12, 2010.
10. Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.
11. Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Scalable concurrent counting. *ACM Trans. Comput. Syst.*, 13(4):343–364, 1995.
12. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, New York, NY, USA, 2003. ACM.
13. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
14. Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.
15. M.P. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems*, 13(1):123–149, January 1991.
16. R. Jain, D. M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, DEC research, TR-301, 1984.
17. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1):21–65, 1991.

18. Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.
19. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.