CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2985

SPECIAL ISSUE PAPER

Micro-transactions for concurrent data structures

Fadi Meawad[1,*,†], Karthik Iyer[1], Martin Schoeberl[2] and Jan Vitek[1]

[1] *Computer Science Department, Purdue University, West Lafayette, IN, USA*
[2] *Technical University of Denmark, Lyngby, Denmark*

SUMMARY

Transactional memory is a promising technique for enforcing disciplined access to shared data in a multiprocessor system. Transactional memory simplifies the implementation of a variety of concurrent data structures. In this paper, we study the benefits of a modest, real-time aware, hardware implementation of transactional memory that we call *micro-transactions*. In particular, we argue that hardware support for micro-transactions allows us to efficiently implement certain data structures. Those data structures are difficult to realize with the atomic operations provided by stock hardware and provide real-time guarantees for those operations. Our main implementation platform is the Java Optimized Processor system, a field-programmable gate array (FPGA) implementation of the Java virtual machine, optimized for real-time Java. We report on the performance of data structures implemented with locks, atomic instructions, and micro-transactions. Our results suggest that transactional memory is an interesting alternative to traditional concurrency control mechanisms. Copyright © 2012 John Wiley & Sons, Ltd.

Received 19 August 2012; Revised 15 November 2012; Accepted 26 November 2012

KEY WORDS: transactional memory; real-time systems; lock-free queues

1. INTRODUCTION

A key challenge in multicore systems is to have efficient synchronization amongst tasks executing concurrently on the system. The challenge becomes even harder on embedded systems because of their real-time requirements. In a real-time system, every task is associated with a deadline. Some of these deadlines, like for a controller of the airbags in a car, have to be met, whereas others may be occasionally missed. Nonblocking algorithms are able to ensure the real-time guarantees required by an embedded system.

Lock-free algorithms are nonblocking algorithms with a system-wide progress guarantee. Dedicated lock-free data structures have been proposed in the literature [1–3]. These algorithms are usually based on dedicated hardware instructions such as compare-and-swap (CAS). The CAS atomic operation instruction is available in current multiprocessor systems. The main limitation of this operation is that it operates on a single memory location, whereas some algorithms require a multiword atomic operation [3, 4]. Although many designs have been proposed for multiword CAS (MCAS), they are not available in commodity hardware. Transactional memory (TM) is an alternative concurrency control mechanism that attempts to simplify parallel programming. A transaction is a nonblocking, serializable group of operations performed atomically. A transaction modifies shared-data structures regardless of the other threads, while keeping a log of the read/write operations. Upon commit (at the end of the transaction), all read operations are checked against writes from other threads. In case of a conflict between transactions, one transaction is

[*] Correspondence to: Fadi Meawad, Computer Science Department, Purdue University, West Lafayette, IN 47906, USA.
[†] E-mail: fmeawad@cs.purdue.edu

aborted and its memory modifications are undone. We employed different variants concerning how eagerly the checks are made, how the read/write logs are maintained, or how the caches are restored.

Transactional memory has been explored both in hardware (HTM) [5–7] and in software (STM) [8, 9]. An HTM implementation handles the logging and the checks in the hardware, which is usually faster but not scalable. Most HTM systems are based on cache coherence protocols to detect conflicts upon write back. If the transaction scales beyond the cache size, all transaction blocks will be retrieved back to the cache for checking. An STM implementation does not require any hardware support except for atomic operations. It performs all the logging and checks in software, allowing very large transactions at the expense of the software overhead. For some problems, TM can provide a straightforward programming model that can provide fine-grained synchronization. However, both STM and HTM come with their problems. STM systems have not been able to exhibit acceptable performance, and HTM requires programmers to be aware of cache sizes.

The purpose of this paper is to show that HTM can serve as a basis for implementing nonblocking algorithms in real-time systems. Nonblocking data structures are currently limited to a subset of the data structures that can be implemented using single-word atomic operations. Their implementation depends on translating a fine-grained locking implementation into a nonblocking implementation by replacing the locks with an atomic operation. If the lock is modifying more than a single memory location at a time, then a direct transformation fails. An example is the attempt to atomically modify the `next` and `tail` pointers in a singly linked queue. Some techniques can be used for memory locations [10], which depend on modifying one, then the other, but that only works for pointers. Other techniques sacrifice the consistency of some of the pointers and restore them when they are being accessed [11, 12], causing an extra overhead on the access of those pointers. Not all data structures can be coded using those techniques. As the complexity of the structure increases, the number of memory locations that needs to be updated atomically also increases.

The main issue is the inability to modify multiple values atomically in a nonblocking manner without an MCAS operation. Transactions have the ability to do that, because each transaction can perform a set of operations atomically. The size of the transaction depends on the number of words intended for the MCAS, which can vary from 2 to 11 depending on the algorithm. For such small transactions, in an STM system, the logging/checking overhead will be dominant, and it will require at least double that number of memory read and writes. HTM on the other hand is a perfect candidate for this situation. Because the transactions are small, the logs will be maintained in the cache or in special registers, and all the checks will be performed by the hardware. We call those small transactions *micro-transactions*.

In this paper, we show efficient lock-free implementations of concurrent first-in, first-out (FIFO) bounded and double-ended dynamically growing queues based on micro-transactions and compare them with lock implementations. We also compare with CAS implementations or explain why CAS is infeasible in some situations. Similar to Schoeberl *et al.* [13], bounding the number of retries on each transaction allows us to guarantee that all tasks will meet their deadlines. For example, if four identical tasks are running concurrently with a single atomic section each, and if the non-atomic section for each task is three times the size of the atomic section, then each transaction will abort three times at the most. If we can guarantee that all tasks will meet their deadlines, then we guarantee that our system satisfies the real-time requirements. We evaluate the performance of the queue implementations against their lock counterparts, while describing why CAS is infeasible or inefficient in each case. For our experiments, we use the real-time TM (RTTM) infrastructure available on the Java Optimized Processor (JOP) [14], an FPGA implementation of a multicore Java processor system.

To evaluate the scalability of our work, we ran our experiments on an Azul (Sunnyvale, CA, USA) machine using a large number of cores. The Azul processor has a runtime feature called speculative multi-address atomicity (SMA) that attempts to run small synchronized blocks transactionally. Because our implementations use micro-transactions, we wrap those transactions with locks, and the Azul virtual machine (VM) restores them to transactions. For the Azul system, we cannot provide any real-time guarantee, but we are able to measure the behavior of nonblocking data structures on a large number of cores.

DOI: 10.1002/cpe

Our experiments show that the TM-based lock-free queue implementations perform better than lock-based queue implementation when contention increases and atomic sections grow in size, making TM an ideal synchronization platform for lock-free algorithms.

This paper is an extension of our previous work [15] and is organized as follows. Section 2 presents background and motivation for our work. Section 3 introduces JOP and RTTM. Implementation of wait-free queues is explained in Section 4. Section 5 presents the experimentation and evaluation of the queues on JOP. Section 6 describes experimentation and results on the Azul machine. The paper is concluded in Section 7.

2. BACKGROUND

From the early days of the TM, Herlihy and Moss [7] suggested that lock-free data structures can be implemented using HTM. Because of the delay of a realization of an HTM and the presence of atomic operations (such as CAS), lock-free data structures using atomic operations emerged. Work on lock-free queues using atomic operations started a couple of decades ago [16, 17]. The Michael and Scott FIFO queue [10] is considered efficient and scalable with two CAS operations for node insertions and one for node removal. Insertions can be reduced to a single CAS [11] by maintaining a doubly linked structure and reversing the direction of insert/remove operations. However, this comes at the cost of occasional O(n) queue traversals to patch inconsistencies, which is an issue for providing real-time guarantees.

Kogan and Petrank [18] proposed a wait-free implementation where higher priority threads help the lower priority peers to complete execution. However, their solution is designed for singly linked list-based queues. Most CAS-based wait-free queues are unbounded. Bounded queues require two atomic operations, one for the queue's end and the other for its size.

We did not find literature on nonblocking implementations of dynamically growing bounded queues. Many other nonblocking algorithms and data structures such as concurrent hash tables and graph structures need MCAS [3, 4]. Bounded wait-free queues are usually implemented using locks and are optimized either for reads or for writes. A `WaitFreeReadQueue` has the write operations guarded with locks, whereas the read operation is wait-free. A `WaitFreeWriteQueue` has the read operations guarded with locks, whereas the write operation is wait-free [19]. The latter implementation is offered as part of the Real-Time Specification of Java [20]. Sundell and Tsigas [12] provided a lock-free implementation of deques using CAS, but their implementation requires fixing the queue in case of inconsistencies.

3. THE HARDWARE PLATFORM JOP

The JOP is a hardware implementation of the Java Virtual Machine [21]. It is a time-predictable bytecode processor with real-time garbage collection (GC) capabilities. JOP is optimized to support static worst-case execution time (WCET) analysis [22]. It is a reduced instruction-set computing-based stack computer that is capable of dynamically translating Java bytecodes into a stack-based *microcode*. It features a four-stage pipeline with the first stage performing the bytecode to microcode translation. The following pipeline stages are microcode fetch, decode and stack address generation, and execute. There are no pipeline dependencies, which simplifies the low-level timing model for the WCET analysis. Memory load and store instructions use an explicit microcode `wait` instruction to avoid sharing the state of the memory controller between bytecodes. JOP features a time-predictable instruction cache and stack caches allowing accurate low-level WCET analysis. JOP is implemented as a soft-core in an FPGA. The time-predictable properties and the availability of RTTM infrastructure made JOP the processor of choice for our experiments. GC support for the multicore version of JOP is under development [23], so we did not rely on a GC for the queue implementations. All the experiments were designed to fit in memory with no GC; we reused the objects to extend the size of the experiment.

We used the multicore version of JOP [24] with four JOP cores for our experiments. JOP uses a preemptive scheduler with fixed priorities. Each thread has a unique priority to avoid managing a

DOI: 10.1002/cpe

FIFO list for each priority level. On a multicore version of JOP, each core executes its own scheduler and threads are bound to individual cores.

3.1. RTTM on JOP

Real-time TM on JOP [14] is a time-predictable hardware implementation of TM that aims at low WCET instead of a high average-case throughput. It supports small atomic sections in concurrent threads with a few read and write operations. The RTTM infrastructure contains a fully associative buffer, local for each core, that caches changed (write operations) data during a transaction. A set of tag memories (local to the core) maintain the read locations (the read data is not cached). The processor state is saved before starting a transaction. On a commit, the changed data in the local cache is copied atomically to the shared memory. A global lock ensures the atomicity of the commit. A conflict is said to occur if the read set of one transaction interferes with the write set of another transaction.

Conflict detection happens only during the commit when all $n - 1$ cores (on an n-core multiprocessor) listen to the core that commits the transaction and not during local read/writes, thus saving valuable CPU cycles. A committing transaction will finish its commit. The other, listening, transactions will abort and restart when a conflict is detected.

We assume a multithreaded real-time application that consists of periodic threads. Within each period, a thread executes a bounded number of transactions. A thread executing a transaction is not preempted by a thread on the same processor core. Threads running on different cores may execute conflicting transactions. With this model of computation, minimal time distances between transactions can be guaranteed and therefore the maximum number of transaction retries bounded. The real-time behavior of such transactions is established by bounding the number of retries r to $n-1$ on an n-core multiprocessor. With threads that execute a single transaction per period and that have periods longer than the conflict resolution time, the maximum number of retries r is $m-1$ for m conflicting transactions, where m can be higher than n when threads are switched during transaction resolution. Additional transactions within a single period can be modeled by additional tasks. The proof can be found in our previous work [13].

Assuming periodic threads, non-overlapping periods and execution deadline not exceeding the period, the WCET of any thread τ is given by the equation

$$\mathrm{WCET} = t_{\mathrm{na}} + (r + 1)t_{\mathrm{amax}} \qquad (1)$$

where WCET is the worst-case execution time, t_{na} is the execution time of the non-atomic section of the thread, and t_{amax} is the maximum of the execution times of the atomic sections of all the n threads in the system. Because r is bounded, the WCET of any thread is bounded.

Atomic sections, the transactions, are represented by methods. The methods are annotated with `@atomic` to mark atomic methods. The JOPizer tool manipulates the atomic methods to implement saving of the local state (the method arguments) and the repeating loop for the restart, and emits code to start and stop a transaction. An transaction abort from the RTTM hardware is signaled via an interrupt, which itself is mapped to a Java exception. That exception is caught, local state reloaded, and the transaction restarted.

4. IMPLEMENTATION

We have implemented dynamically growing bounded FIFO queues with supported primitives *insert at tail* and *remove from head*. We have also implemented doubly linked double-ended queue that supports the primitives *insert at head* and *remove from tail* together with the ones mentioned earlier. We have simulated a stack using the double-ended queues by inserting and removing elements from the same end of the queue. The queue implementations are concurrent, and synchronization is achieved using two different synchronization techniques, locks and TM. The different variants are explained in the following paragraphs. In our implementations, we use the Java `synchronized` keyword to implement locks and the annotation `@atomic` for micro-transactions.

DOI: 10.1002/cpe

*4.1. Bounded queue (*bounded queue*)*

The *bounded queue* is a doubly linked queue with capacity constraints. This requires maintaining the queue size as well as checking it against the capacity during insertions for a queue-full condition. Also, the insertion and removal primitives now have to atomically increase and decrease the queue size. This problem is usually solved by using a multiword CAS primitive or by limiting the queue to a single reader or a single writer [19]. Hence, we do not present implementations for the CAS variants. We implement only the lock and the TM variants. Figure 1 summarizes the implementation. The queue insertion and removal methods are protected either by usage of `synchronized` (as described in Figure 1) or by replacing the `synchronized` keyword with `@atomic` for the TM alternative.

*4.2. Double-ended queue (*deque*)*

deque is a double-linked queue, which allows insertion or removal from either side. Unlike the bounded queue, the deque has a CAS implementation, but the CAS implementation is inefficient.

```
final static class Node {
  Object value;
  volatile Node next;
  volatile Node previous;
}
volatile Node head = new Node();
volatile Node tail = new Node();
final int capacity;
volatile int size;
synchronized boolean insert_tr(Node n) {
  if (size < capacity) {
    n.previous = tail.previous;
    n.next = tail;
    tail.previous.next = n;
    tail.previous = n;
    size++;
    return true;
  }
  return false;
}
synchronized Node remove_tr() {
  if (head.next != tail) {
    Node n = head.next;
    head.next = n.next;
    n.next.previous = head;
    size--;
    return n;
  }
  return null;
}
boolean insert(Object o) {
  Node n = new Node();
  n.value = o;
  return insert_tr(n);
}
Object remove() {
  Node n = remove_tr();
  if (n != null)
    return n.value;
  return null;
}
```

Figure 1. Dynamically growing *bounded queue*.

The reason for the inefficiency is that the implementation maintains consistency only of the `next` pointer, and the `previous` pointer is adjusted in O(n) operations when used. The CAS implementation of the deque follows the Java class `ConcurrentLinkedDeque`,[‡] which is a concurrent, doubly linked, double-ended queue. The implementation is based on the technique described by Martin [25]. The queue has separate `head` and `tail` nodes, and newly inserted nodes are either the immediate successor of the `head` or the immediate predecessor of the `tail`. Our implementation provides primitives to insert and remove both at the head and the tail. The CAS implementation of deque assumes that queue inconsistencies happen rarely, and when they do, they are corrected when required. In this implementation, the consistency of the `next` pointers of the queue nodes is ensured by using CAS instructions, whereas the `previous` pointers are updated optimistically using store operations and may not be updated correctly. When necessary, the `previous` pointers have to be reconstructed using the `next` pointers. Also, when nodes are deleted, node pointers may not be updated correctly, resulting in valid nodes pointing to deleted nodes. When necessary, the node pointers have to be updated to point to valid nodes. Some queue primitives require that these inconsistencies be corrected and hence invoke correction routines. The following summarizes the primitives:

- *Insert Head*: The new node `X` is inserted as a successor of the queue `head`. Pointer `X.next` is updated to `head.next` (say `N`) using a CAS instruction only if the node `head.next` is not already deleted. Pointers `X.previous` and `N.previous` are updated optimistically using a store instruction to `head` and `X`, respectively. Note that this operation does not need to reconstruct the `previous` pointers.
- *Insert Tail*: The new node `X` is inserted as a predecessor of the queue `tail`. This operation requires that the pointer `tail.previous` points to a valid predecessor `P` to update `P`'s `next` pointers. Because the `previous` pointers are optimistically updated using regular store instructions, a correction routine is invoked to reconstruct the pointers before the node is inserted. The `next` pointers are updated using CAS instructions, whereas the `previous` pointers are updated optimistically.
- *Remove Head*: This operation removes a node that is an immediate successor of the `head`. This requires that we find a valid (not already deleted) node `X` that is a successor of the `head`. A correction routine is invoked that starts from the `head` and skips all deleted nodes to identify the true successor of the `head`, node `X`. Again, the `next` pointers are updated atomically and the `previous` pointers optimistically.
- *Remove Tail*: This operation removes the immediate predecessor of the `tail`. As in the case of *Insert Tail*, a valid predecessor `P` of the `tail` has to be identified before its removal, requiring an invocation of the `previous` pointer reconstruction routines.

On the other hand, the code for the lock and TM implementation is much simpler and straightforward; both `previous` and `next` pointers are updated together atomically. The listing in Figure 2 contains the implementation of the deque. In the case of TM, the methods `takefirst_tr`, `takelast_tr`, `putfirst_tr`, and `putlast_tr` are annotated with `@atomic` instead of the `synchronized` keyword that is used for the lock implementation.

The number of CAS instructions required per operation is not constant for the latter three operations as an arbitrary number of pointers may have to be atomically corrected when the reconstruction routines (`previous` pointer correction and removing links to already deleted nodes) are invoked. The reconstruction routines have a significant impact on the performance. JOP currently does not provide native support for CAS. Therefore, each CAS will be simulated in our experiments with a transaction. JOP transactions have very small overhead. That overhead is negligible compared with the inefficiency of the algorithm itself. The lock and TM implementations of the deque are simple and straightforward. To keep the lock and TM implementations consistent with the CAS versions, the deque implementations have separate `head` and `tail` nodes, and insertions and removals operate on the successor of the `head` and predecessor of the `tail`, respectively. They provide

[‡] http://g.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/jsr166x/ConcurrentLinkedDeque.java?view=co

```
final static class Node {
 Object value;
 volatile Node next;
 volatile Node previous;
}
volatile Node head = new Node();
volatile Node tail = new Node();
synchronized Node takefirst_tr() {
 Node n = null;
 Node h = head;
 if (h.next != tail) {
  n = h.next;
  h.next = n.next;
  n.next.previous = h;
 }
 return n;
}
synchronized Node takelast_tr() {
 Node n = null;
 if (head.next != tail) {
  n = tail.previous;
  tail.previous = n.previous;
  n.previous.next = tail;
 }
 return n;
}
synchronized void putfirst_tr(Node n) {
 Node h = null;
 h = head;
 n.previous = head;
 n.next = h.next;
 h.next.previous = n;
 h.next = n;
}
synchronized void putlast_tr(Node n) {
 n.previous = tail.previous;
 n.next = tail;
 tail.previous.next = n;
 tail.previous = n;
}
void putfirst(Object o) {
 Node n = new Node();
 n.value = o;
 putfirst_tr(n);
}
void putlast(Object o) {
 Node n = new Node();
 n.value = o;
 putlasr_tr(n);
}
Object takefirst(){
 Node n = takefirst_tr();
 if (n != null)
  return n.value;
 return null;
}
Object takelast(){
 Node n = takelast_tr();
 if (n != null)
  return n.value;
 return null;
}
```

Figure 2. Dynamically growing *deque*.

routines to insert and remove from both ends of the queue. As with other queues, the insert and remove primitives are associated with `synchronized` or `@atomic`.

4.3. *Stack (*stack*)*

The *stack* is simulated using the deque by inserting and removing queue elements from a common end of the queue. In our experiments, stack increases contention as multiple threads access the same end of the queue. Push operation uses either *Insert Head* or *Insert Tail*, and the pop operation uses *Remove Head* or *Remove Tail*, respectively. All threads agree on the same end.

5. EXPERIMENTATION AND EVALUATION

The experimentation environment is an FPGA programmed with a symmetric shared-memory multiprocessor hardware system with four JOP cores. For the hardware platform, we use an Altera DE2-70 Development board[§] consisting of a Cyclone II EP2C70 FPGA (San Jose, CA, USA). The Altera board contains 64-MB SDRAM, 2-MB SSRAM, and an 8-MB Flash memory and I/O interfaces such as a USB 2.0, a RS232, and a ByteBlasterMV port. Each JOP core has a core-local 4-KB instruction cache and 1-KB stack cache. The Cyclone FPGA is programmed to implement a symmetric shared-memory multiprocessor environment. To evaluate and compare the various synchronization primitives for concurrent lock-free queues, we conducted experiments using a producer–consumer example on a four-core symmetric multicore system, each core executing Java bytecode. Each of the four cores executed an independent shared-memory thread with one of producer-only or consumer-only functionality. The queue nodes were exchanged among the concurrent threads rendering the queue `head` and `tail` pointers and nodes contention points. Synchronization is achieved through lock and TM primitives. We collected and compared the execution times and other TM properties. The rest of the section explains the framework and the evaluation in detail.

5.1. *Producer–consumer example*

The producer and the consumer functionality are performed by independent JOP threads referred to as inserter and remover threads, each executing on a separate JOP core. The producer, in a loop, inserts `num` queue nodes into `Q`, and similarly, the remover removes `num` nodes from `Q`. We measured our experiments for four different variations; The first variation is single producer-single consumer (1-1), allowing us to measure any overhead the TM system incurs as well as establishing a baseline for the performance. The second and third variations are single producer-two consumers (1-2) and two producers-single consumer (2-1), respectively. The purpose of these variations is to create contention on only one of the two ends of the queue. And finally, with two producers-two consumers (2-2), we create contention on both ends of the queue. In case there are more than one producer or consumer, the number of nodes inserted or removed are divided equally among the threads.

The queue insertion and removal operations are performed atomically using the synchronization variants mentioned earlier.

All required synchronization is included in the queue methods; neither the producer nor the consumer needs or has any extra synchronization. Such a system of threads, composed of atomic and non-atomic sections, conforms with the thread model used to establish real-time bounds on the number of retries, as in Schoeberl *et al*. [13]. In each experiment, we started all the threads simultaneously. The contention points in such an experimental setup are the `head` and `tail` pointers of the queue and the pointers associated with the queue nodes exchanged among various threads.

5.2. *Experimentation*

We conducted experiments to record the execution time, number of commits and retries, and the size of read and write sets of the different queue implementations.

[§] http://www.altera.com/education/univ/materials/boards/de2-70/unv-de2-70-board.html

DOI: 10.1002/cpe

Table I. List of all experiments.

	Insert at	Remove from
Bounded queue	Head	Tail
Double-ended queue	Head	Tail
	Head	Random
Stack	Head	Head
	Tail	Tail

- bounded queue: In the bounded queue, for different capacities, we insert at one end (`head`) and remove from the other end (`tail`). If capacity is reached, the thread spins until a spot is available on the queue.
- deque: For the deque, we experimented many variations of insertions and deletions, such as inserting at (`tail`) and removing from (`head`). We have also added a randomness factor, that is, the insertion or removal can happen on either end of the queue randomly.
- stack: Using the deque, a stack is simulated by having all operations performed on one end; the experiment is performed on each end.

For the evaluation of TM synchronization techniques, we are interested in the total number of commits and retries, and the maximum size read/write sets of an experiment. The effect of workload and transactional data size on the number of commits/retries and the read/write sets of an individual core has been dealt in detail by Schoeberl *et al*. [13]. Table I lists the different implementations and their variations that we used in this paper. The first column is the queue type, and the second and third columns present the side of the queue where the queue insertion or removal is performed, respectively. In bounded queue, only one configuration is feasible. For stack, both possible configurations are covered. We have selected a representative subset of the possible combinations for the deque. The deque and stack experiments were also conducted using the simulated CAS.

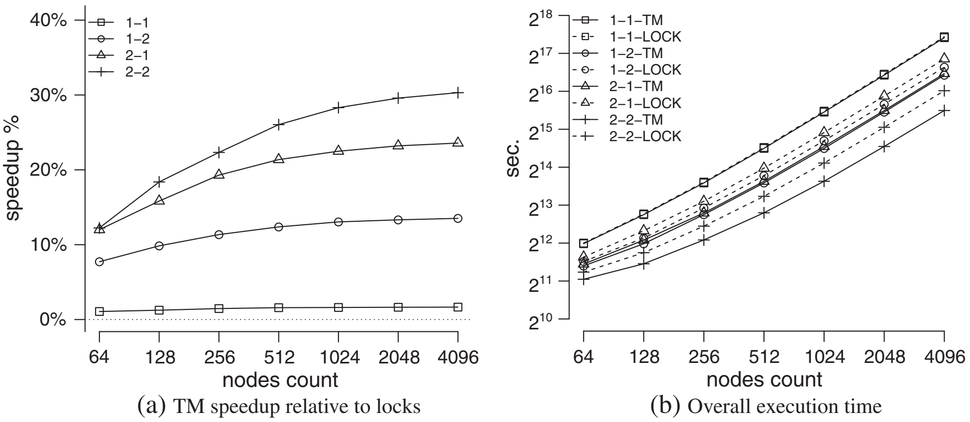
5.3. Evaluation

The graphs shown in this section are based on the average of three runs; the σ for all nonrandom experiments is less that 3 ms. The difference between two configurations at any data point is more than 50 ms.

5.3.1. Bounded queue. In our implementation, a bounded queue is a doubly linked queue with limited capacity. Such queues increase contention by forcing three operations, a check for queue full, an increment of the current queue size, and the actual node insertion, to be executed in a single atomic step. Experiments on the bounded queue were conducted using only the lock and TM variants as, according to our knowledge, there is no single-word CAS-based bounded queue implementation.

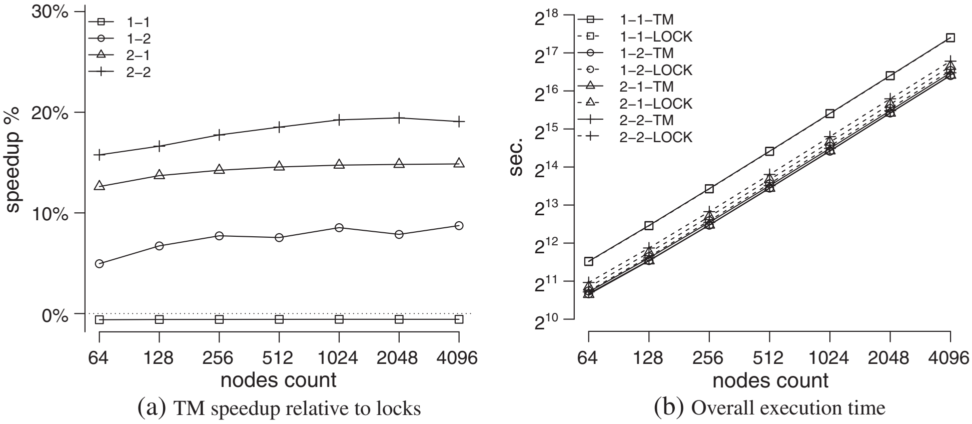
Figure 3 shows the speedup percentage of the different implementations and configurations, each compared to its equivalent locking counterpart. The single producer-single consumer lock implementation is the slowest, not because of contention but due to low production/consumption rate. At only 1% speedup from the 1-1 lock implementation, the corresponding TM implementation comes as the second slowest for that experiment. With 30% speedup, the 2-2 TM implementation is the fastest configuration. Locks are held longer by threads because of the additional check and increment operations while inserting and a decrement operation during removal. This significantly increases the execution time. The 1-2 lock configuration is 13% slower than the 2-1 lock implementation, whereas both TM configurations behaved comparably. Bounded queue suffers up to 13.5% retries per operation ratio when there is more than one thread contending at the tail (for the 1-2 and 2-2 cases). There is no similar contention on the producer side, as it is more likely that both consumers will be pending on an item to be removed from `Q` more than two producers waiting for a spot in `Q` to fill.

DOI: 10.1002/cpe



(a) TM speedup relative to locks (b) Overall execution time

Figure 3. *Bounded queue* for transactional memory (TM) versus lock on Java Optimized Processor. (a) TM speedup relative to locks and (b) overall execution time.



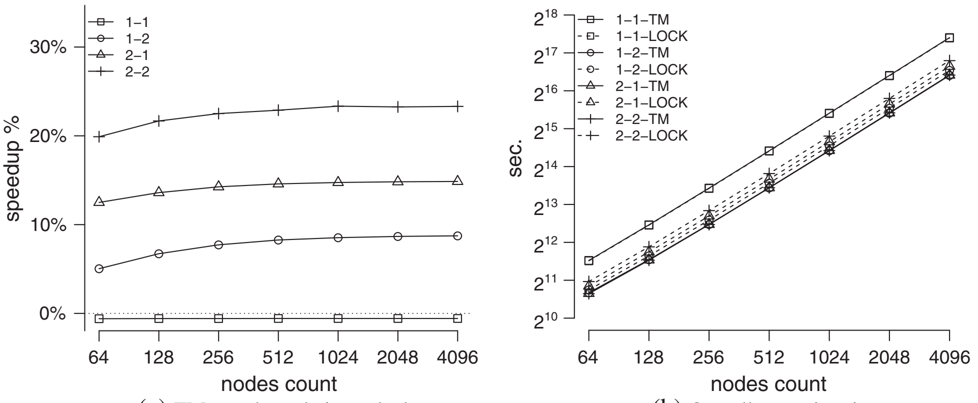
(a) TM speedup relative to locks (b) Overall execution time

Figure 4. Head–tail configuration for transactional memory (TM) versus lock on Java Optimized Processor. (a) TM speedup relative to locks and (b) overall execution time.

5.3.2. deque. As mentioned earlier, we conducted two different experiments on double-ended queues that varied in the queue ends at which insertions/removals were executed. In Figure 4, the deque is used as an FIFO queue, insertions at the head and removal from the tail. With up to 20% speedup, the 2-2 TM implementation outperformed the lock implementation. In the head–tail 1-1 configuration, the TM implementation suffers ∼1% slowdown. The difference in speedup between the 1-2, 2-1 and 2-2 configurations is due to different contention loads on each. The 1-2 configuration has the least contention, because the consumers are contending on an empty queue, which results in small a contention (only the shared pointers are read to check if the queue is empty). Whereas in the 2-1 configuration, the contention is more expensive as there is no check resulting in a retry instead (which is better than busy waiting). In the 2-2 configuration, we observe a contention similar to the 2-1 configuration contention on both ends, magnifying the effect. This might be because a consumer waiting for a single producer takes less time than a consumer TM retry, or it might be just noise. Similar results where achieved when randomly selecting the consumer end of the queue at each consumption. The speedup for the random configuration is shown in Figure 5.

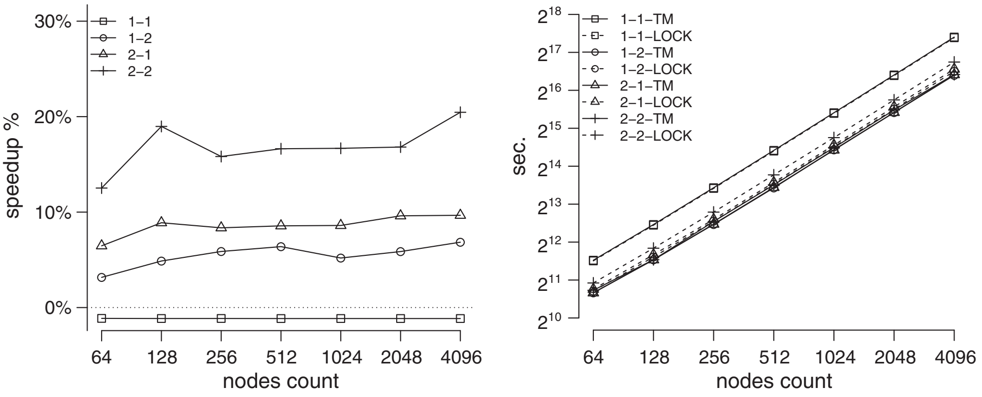
5.3.3. Stack. As discussed, a stack is simulated by inserting and removing queue nodes from a common queue end. This creates more contention as all the threads operate on either the queue `head` and its successor or the `tail` and its predecessor. The results of the stack-based experiments are similar in nature to its deque counterparts. The speedups for the stack configurations are shown in Figures 6 and 7.

DOI: 10.1002/cpe



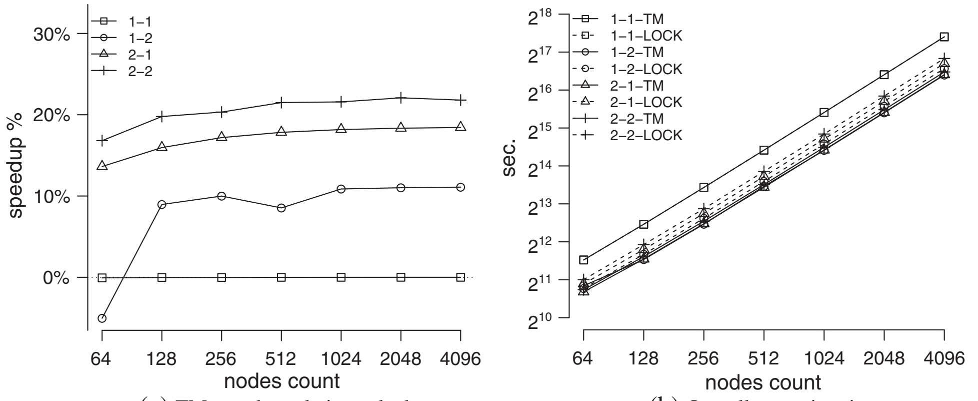
(a) TM speedup relative to locks (b) Overall execution time

Figure 5. Head–random configuration for transactional memory (TM) versus lock on Java Optimized Processor. a) TM speedup relative to locks and (b) overall execution time.



(a) TM speedup relative to locks (b) Overall execution time

Figure 6. *Stack* (simulated with *deque*, insertions, and deletions at the head) for transactional memory (TM) versus lock on Java Optimized Processor. (a) TM speedup relative to locks and (b) overall execution time.



(a) TM speedup relative to locks (b) Overall execution time

Figure 7. *Stack* (simulated with *deque*, insertions, and deletions at the tail) for transactional memory (TM) versus lock on Java Optimized Processor. (a) TM speedup relative to locks and (b) overall execution time.

DOI: 10.1002/cpe

5.3.4. Comparing with CAS. Unlike bounded queue, deque has a CAS implementation, but we argue that it is marginally slower than both TM and lock implementations. Because JOP does not have a native CAS instruction, we simulated the CAS instruction with a transaction. The added overhead of the simulation in this experiment is negligible compared with the time spent in reconstructing the `previous` pointers. For the same reason, unlike the lock and TM implementations, the end from where the insertions and removals are performed affects the performance of the CAS implementation largely. Figure 8 shows the CAS implementation slowdown for the different deque configurations of a single producer-single consumer compared with the head–tail lock configuration. It is worth noting that all other TM and lock configurations are within 1.5% of the baseline. Increasing the number of threads increases the CAS implementation slowdown. The CAS tail–head configuration is almost twice as slow as the CAS head–tail configuration due to running the correction routines.

5.3.5. Transactional read–write sets. Table II shows the sizes of the read sets, write sets, and the union of the read and write sets for the TM queues. The bounded queue is implemented as a doubly linked list with a *capacity* parameter, which is reflected in its read, write, and read–write set. The deque and stack have bigger sets as more shared pointers are read/modified per operation. As can be seen, the size to the read and write sets fits very well for the concept of micro-transactions. This data can be held with reasonable hardware effort in fully associative buffers.

The read and write set sizes for the CAS-based queues do not vary. The deque and stack data structures use significantly more CAS instructions than a singly or a doubly linked queue. Irrespective of the number of CAS instructions executed by the insertion and removal operations, the transaction size of each CAS instruction is constant across all queues/stack as a CAS instruction reads and modifies only a single pointer. Hence, the size of the read and write sets is the same for all queue variants and stack.

5.3.6. Commits and retries. TM and CAS both depend on retires of failing commits. The CAS-based implementation records more commits than the TM versions because CAS implementations

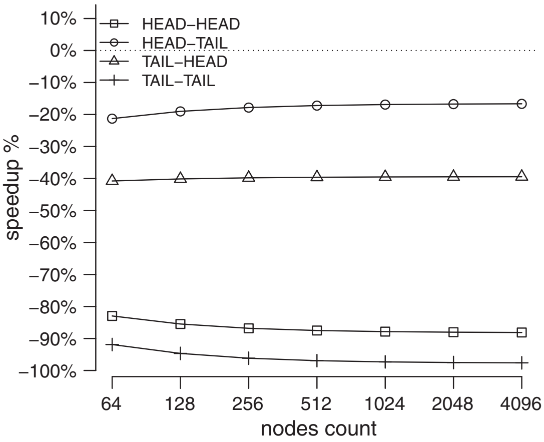


Figure 8. Single producer-single consumer, compare-and-swap *deque* % slowdown versus lock head–tail deque on Java Optimized Processor.

Table II. Read, write, and read–write sets of TM-based queues.

		Read set	Write set	Read–write set
TM	Bounded	7	2	7
	Double-ended	11	4	12
	Stack	11	5	12

TM, transactional memory.

Table III. Retries.

	Bounded queue (%)	*deque*		*Stack*	
		Head–tail (%)	Head–random (%)	Head (%)	Tail (%)
1-1	0.00	0.22	48.06	0.44	0.22
1-2	12.06	16.87	49.57	4.93	5.74
2-1	0.44	0.22	17.60	10.93	0.89
2-2	13.33	27.96	34.80	28.89	28.78

involve more transactions than the corresponding TM implementations. For example, the double-ended queue implementation requires several CAS instructions to insert/remove nodes from a doubly linked list. In the case of TM, an increase in queue node pointers will not increase the number of commits because all the necessary pointer modifications are performed atomically by a single transaction. However, as the number of processed nodes increases, the number of commits increases. As the number of transactions increases, there is a high probability that the number of retries also increases as transactions being committed may conflict with other transactions.

It can be noted from the figures that the TM-based queues perform better than their lock and CAS-based counterparts, even under high contention, and the performance of TM does not degrade on the basis of the choice of queue ends to insert and removal in most cases. Table III indicates the average number of retries per queue operation for bounded queue, deque, and stack. The average is calculated by dividing the total retries by the total number of queue operations (two operations per queue node processed). In most cases, the retries increase with the increase of contention. The 1-2 configuration does not follow that rule, as it has the same amount of contention as 2-1, but it has a much higher retries count. The retries in the 1-2 case are shorter (detected faster), but the contention is the same, resulting in more retries.

5.4. *Other experiments*

We conducted other experiments on the various implementations of the queues. Three of these are worth noting:

1. Change in capacity of a limited capacity queue: We conducted experiments on the limited capacity queues by varying the capacity of the queue for a specified number of queue nodes and measuring the execution time, retries, and commits. We did not notice any change in the results with the change in queue capacity. We noticed a slight increase in the execution time when the capacity is lowered to very small values. This is due to threads finding that the queue is full. And we have noticed that the slowdown is consistent among all implementations.
2. We also conducted some experiments that introduced a *mover*. A mover consumes from a queue and produces what was consumed to another queue. We experimented having a single mover or two concurrent movers. Again, the results did not expose any different behavior from the one described earlier and were therefore omitted.
3. We tested more configurations of deque, like random–random, random–tail, and tail–head. The results did not differ from what is presented here in the paper.

5.5. *Summary*

Bounded queue, deque, and stack test the TM infrastructure and compare it with other synchronization methods with increasing degrees of contention.

When contention and read/write set sizes increase as in the case of bounded queue, deque, and stack, TM-based implementations outperform the lock-based implementations. Note that the TM-based implementations make dynamically growing bounded buffer lock-free queues feasible, which otherwise need multiword CAS instructions. Figure 3 shows that lock-based bounded buffer queues are quite expensive. In our experiments, the TM-based implementation performs better than locking by up to 30%. It is also noted that TM performance does not vary depending on the

DOI: 10.1002/cpe

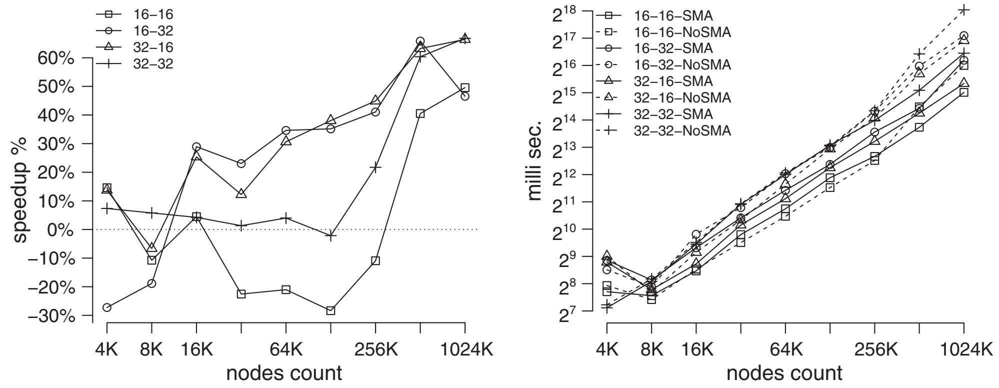
type of queue operations chosen unlike CAS-based implementations. The stack-based experiments simulated the highest degree of contention with operations performed on the same end on a single queue. Figures 6 and 7 suggest that the TM implementations outperform the lock implementations by over than 20% and are significantly faster than CAS-based implementations.

6. EXPERIMENTS ON THE AZUL SYSTEM

We carried out the experiments described in Section 5 on the Azul machine, but with a higher number of threads. The Azul environment does not provide a facility to explicitly specify the intention to use micro-transaction support. Instead, it offers a runtime flag called SMA. With this flag set, the Azul runtime attempts to run the synchronized parts (marked with the `synchronized` keyword) transactionally using hardware transactional support. Azul is known to perform well for small transactions. Unlike experimenting with JOP, because we do not have enough information on how SMA works in Azul nor the number of retries for each transaction, we cannot provide any guarantee on the real-time behavior of the algorithms. But it is interesting to observe experimentally that the algorithm scales up to 64 cores. We experimented only with the lock and the TM versions both with and without SMA support. In the rest of the text, NoSMA indicates that the SMA property is disabled. We carried out experiments starting from 4K (K = 1024) queue nodes to 1024K nodes. The Azul machine is an Azul Vega 3 3310B, with two 54-core processors and 48 GB of RAM. The benchmark ran on top of the Azul Virtual Machine with the Concurrent Pauseless GC [26]. The results discussed in these graphs are based on the average of 10 runs. Although the numbers are consistent across the different runs, we have noticed a strange behavior at 16K nodes experiments that we cannot explain (Azul is a black box with no available implementation details). We think it might be an Azul optimization that gets utilized at that special case.

Given that the synchronized code is small in all experiments, the runs with the SMA flag set performed better. Unfortunately, the number of commits and retries is not provided by the Azul runtime for further analysis. Experiments were conducted starting for 4, 8, 16, 32, 64, and 128 total threads with similar results. The results displayed are using 16 producer and 16 consumer threads (16-16), up to 32 producer and 32 consumer threads (32-32). The latter configuration uses a total of 64 threads, which is the maximum number of threads from our test series with no overflow (108 cores available). The following figures are based on the average of three runs with the thread configuration described above.

Figure 9 shows the results of the bounded queue experiments. The SMA implementation for 16-32 and 32-16 is strictly faster than its NoSMA counterpart for sizes ⩾16K. The 32-32 configuration is faster starting from 256K nodes, and the 16-16 is faster starting from 512K nodes. The 16-16 configuration is slower in most cases. Also, increasing the number of nodes operated on



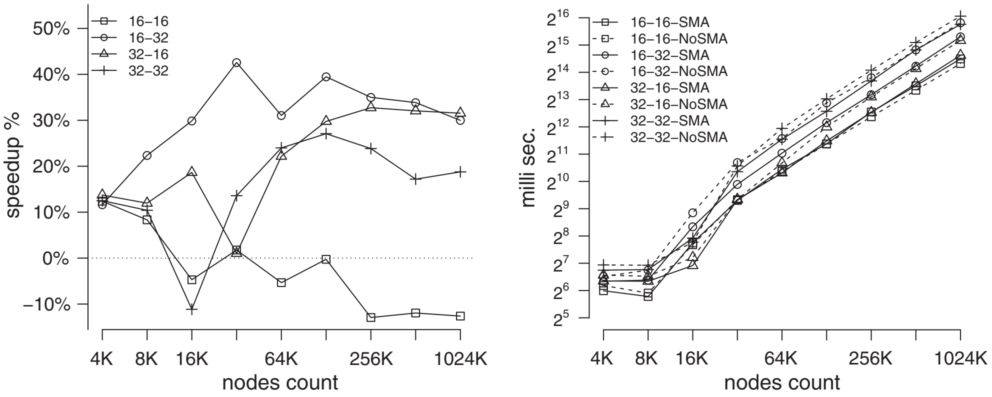
(a) SMA speedup relative to NoSMA

(b) Overall execution time

Figure 9. *Bounded queue* for SMA versus NoSMA on Azul. (a) SMA speedup relative to NoSMA and (b) overall execution time.

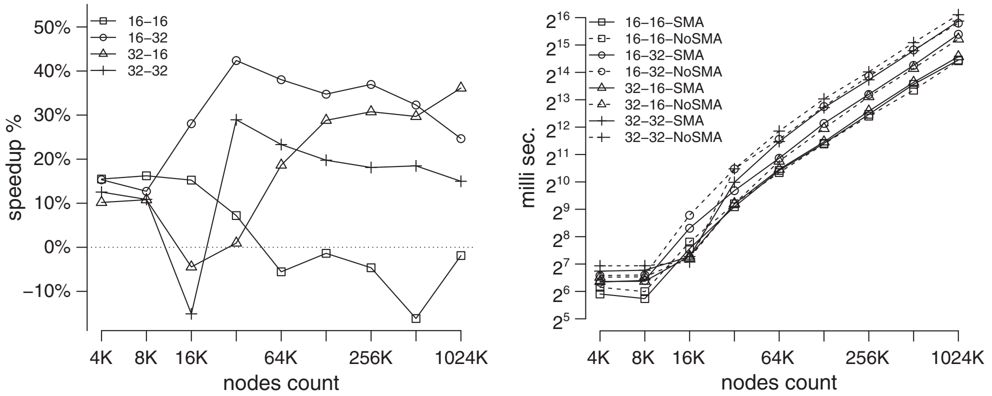
DOI: 10.1002/cpe

yields better performance for the SMA; therefore, it scales over the size of the data as well. It is worth mentioning, that unlike the JOP experiments, where the least number of threads is the slowest because of a slow production cycle, on Azul, the absolute performance is inversely proportional with the thread count because of contention.



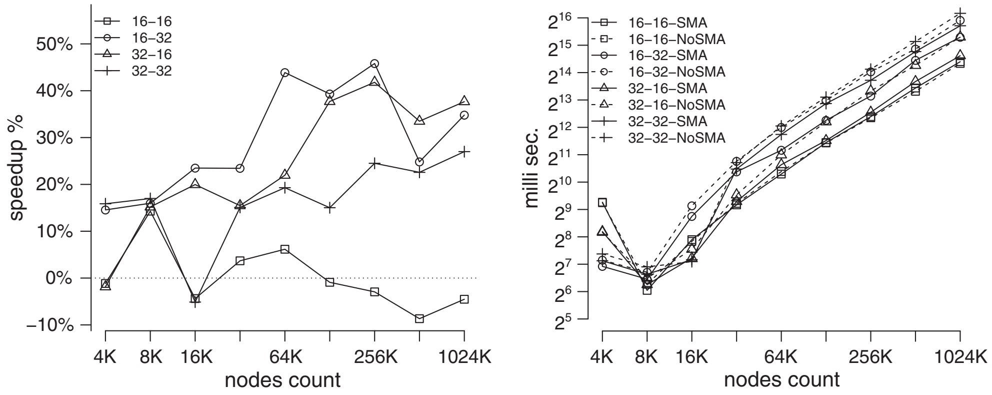
(a) SMA speedup relative to NoSMA (b) Overall execution time

Figure 10. *deque* head–tail configuration for SMA versus NoSMA on Azul.



(a) SMA speedup relative to NoSMA (b) Overall execution time

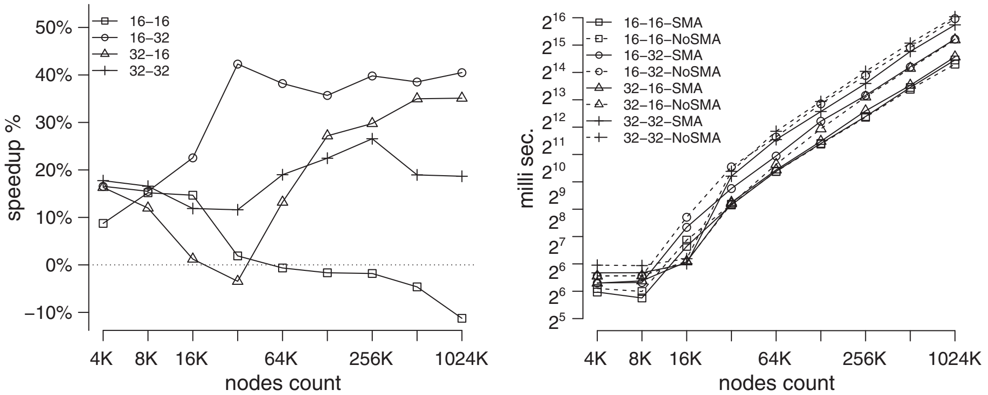
Figure 11. *deque* head–random configuration for SMA versus NoSMA on Azul.



(a) SMA speedup relative to NoSMA (b) Overall execution time

Figure 12. *Stack* (simulated with deque, insertions, and deletions at the head) for SMA versus NoSMA on Azul. (a) SMA speedup relative to NoSMA and (b) overall execution time.

DOI: 10.1002/cpe



(a) SMA speedup relative to NoSMA

(b) Overall execution time

Figure 13. *Stack* (simulated with **deque**, insertions, and deletions at the tail) for SMA versus NoSMA on Azul. (a) SMA speedup relative to NoSMA and (b) overall execution time.

Similarly, we ran the deque implementation on Azul. Figure 10 shows the speedup % for deque head–tail with similar configurations as described in bounded queue. The results are similar as well, except for the 16-16 configuration. In the latter one, SMA is slower than NoSMA, but we expect it to become faster with a larger number of nodes. The same results are observed for the head–random configuration (Figure 11) and the simulated stacks (Figures 12 and 13).

7. CONCLUSION AND FUTURE WORK

This work considers TM as an alternative to CAS and lock-based synchronization primitives. We showed that concurrent algorithms requiring multiword CAS primitives can be implemented using TM in a straightforward way. Our JOP experiments suggest that the TM-based implementations of concurrent nonblocking queue algorithms perform better than the CAS-based implementations. We also showed that as atomic sections and contention grow, the TM-based implementations perform better. The experiments on the Azul platform indicate that TM-based implementations are scalable.

Future work involves exploring more complex data structures, such as graph structures and hash tables. Analyzing TM performance on larger systems with increased contention would also be useful. A WCET analysis would state that in a given period, only a fixed number of transactions is to be executed. Such a requirement raises questions such as which transactions to execute and on what parameters (thread priority, deadlines, etc.) the decision is to be made. Those questions can be explored from a scheduling theory perspective.

ACKNOWLEDGEMENT

This material is based upon the work supported by the National Science Foundation under grant nos. 0958465, 0811691, and 0720652.

REFERENCES

1. Harris T, Fraser K, Pratt IA. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, Toulouse, France, October 2002; 265–279.
2. Herlihy M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 1991; **13**(1):124–149.
3. Ramamurthy S. A lock-free approach to object sharing in real-time systems. *PhD Thesis*, University of North Carolina at Chapel Hill, 1997.
4. Purcell C, Harris T. Non-blocking hashtables with open addressing. *Technical Report UCAM-CL-TR-639*, University of Cambridge, Computer Laboratory, September 2005.

DOI: 10.1002/cpe

5. Ananian CS, Asanović K, Kuszmaul BC, Leiserson CE, Lie S. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, San Francisco, CA, USA, 2005; 316–327.
6. Hammond L, Wong V, Chen M, Carlstrom BD, Davis JD, Hertzberg B, Prabhu MK, Wijaya H, Kozyrakis C, Olukotun K. Transactional memory coherence and consistency. *SIGARCH Computer Architecture News* 2004; **32**:102.
7. Herlihy M, Moss JEB. Transactional memory: architectural support for lock-free data structures. *SIGARCH Computer Architecture News* May 1993; **21**:289–300.
8. Shavit N, Touitou D. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, Ontario, Canada, 1995; 204–213.
9. Spear MF, Marathe VJ, Dalessandro L, Scott ML. Privatization techniques for software transactional memory. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Portland, Oregon, USA, 2007; 338–339.
10. Michael MM, Scott ML. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, USA, 1996; 267–275.
11. Ladan-Mozes E, Shavit N. An optimistic approach to lock-free fifo queues. *Distributed Computing* 2008; **20**(5):323–341.
12. Sundell H, Tsigas P. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing* 2008; **68**(7):1008–1020.
13. Schoeberl M, Brandner F, Vitek J. RTTM: real-time transactional memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, Sierre, Switzerland, 2010; 326–333.
14. Schoeberl M, Hilber P. Design and implementation of real-time transactional memory. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL)*, Milano, Italy, 2010; 379–284.
15. Meawad F, Iyer K, Schoeberl M, Vitek J. Real-time wait-free queues using micro-transactions. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*. ACM: York, UK, September 2011.
16. Treiber RK. Systems programming: coping with parallelism. *Technical Report RJ5118*, IBM Almaden Research Center, April 1986.
17. Valois JD. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, Nevada, 1994; 64–69.
18. Kogan A, Petrank E. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM: New York, NY, 2011; 223–234.
19. Tsigas P, Zhang Y, Cederman D, Dellsen T. Wait-free queue algorithms for the real-time Java specification. In *IEEE Real Time Technology and Applications Symposium*. IEEE Computer Society: Washington, DC, 2006; 373–383.
20. Bollella G, Gosling J, Brosgol B, Dibble P, Furr S, Turnbull M. *The Real-Time Specification for Java*, Java Series. Addison-Wesley: Boston, MA, June 2000.
21. Schoeberl M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture* 2008; **54/1–2**:265–286.
22. Schoeberl M, Puffitsch W, Pedersen RU, Huber B. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience* 2010; **40/6**:507–542.
23. Puffitsch W. Hard real-time garbage collection for a Java chip multi-processor. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11. ACM: New York, NY, USA, 2011; 64–73.
24. Pitter C, Schoeberl M. A real-time Java chip-multiprocessor. *ACM Transactions on Embedded Computing Systems* 2010; **10**(1):9:1–34.
25. Martin PA. Practical lock-free doubly-linked list, Patent, 05 2009. US 7533138.
26. Click C, Tene G, Wolf M. The pauseless GC algorithm. In *International Conference on Virtual Execution Environments (VEE)*, Chicago, IL, 2005; 46–56.

DOI: 10.1002/cpe