Real-Time Wait-Free Queues using Micro-Transactions

Fadi Meawad
Purdue University
fmeawad@cs.purdue.edu

Karthik Iyer
Purdue University
iyer2@cs.purdue.edu

Martin Schoeberl
Technical University of Denmark
masca@imm.dtu.dk

Jan Vitek
Purdue University
jv@cs.purdue.edu

ABSTRACT

This paper evaluates the applicability of transactional memory to the implementation of different non-blocking data structures in the context of the Real-time Specification for Java. In particular, we argue that hardware support for micro-transaction allows us to implement efficiently data structures that are often difficult to realize with the atomic operations provided by stock hardware. Our main implementation platform is the Java Optimized Processor system. We report on the performance of data structures implemented with locks, compare and swap and micro-transactions. Our results confirm that transactional memory is an interesting alternative to traditional concurrency control mechanisms.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming–parallel programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming; H.2.4 [**Systems**]: Transaction Processing

General Terms

Memory, Optimizations, Real-time, Java Processor

Keywords

Transactional Memory, CAS, MCAS, Wait-Free queue

1. INTRODUCTION

Embedded electronic devices have become ubiquitous and an integral part of daily lives. As our expectations in terms of processing power and battery life keep increasing, multi-core systems are being considered for high performance applications, including real-time ones, owing to their low energy and thermal profile [30, 13]. One challenge in programming multi-core, real-time, embedded systems is how to implement efficient synchronization amongst tasks executing concurrently on the system. Dedicated lock-free data structures have been proposed in the literature [5, 6, 16]. These algorithms are usually based on dedicated hardware instruction such as compare and swap (CAS). Although initial hardware realizations were slow, the performance of CAS is now comparable to regular instructions. Also, CAS instructions are fully integrated in current multi-processor systems. The main limitation of CAS is that it operates on a single memory location while some algorithms require a multi-word CAS (MCAS) [15, 16]. Although many designs have been proposed for MCAS, they are not available in commodity hardware.

Transactional Memory (TM) is an alternative synchronization infrastructure that solves some of the problems associated with CAS [7, 11]. Transactions are non-blocking, serializable, atomic read/write operations executed by concurrent threads. Transactions own a part of shared memory and they either complete and commit memory changes or abort and retry if the same area is owned by another transaction. Transactional Memory has been explored both in hardware (HTM) [1, 4, 7] and in software (STM) [22, 24]. Transactional Memory features several advantages like ease of implementation, straight-forward programming model and ability to combine fine and coarse grained operations [11]. However, both STM and HTM come with their problems. STM system have not been able to exhibit acceptable performance and HTM requires programmers to be aware of cache and buffer sizes.

The purpose of this paper is to show that HTM can serve as basis for implementing non-blocking algorithms in real-time systems. We focus on the Real-time Specification for Java [2] and look at the implementation of the wait-free queue classes that are part of the specification. Some of the problems associated with non-blocking CAS based queue implementations can be solved using TM. Ideas that required MCAS, like the wait-free bounded capacity queue can instead utilize an efficient HTM for small transactions, which we call micro-transactions.

In this paper we show efficient wait-free implementations of concurrent FIFO queues based on micro-transactions and compare them with lock and CAS based implementations. Similar to [20], this paper provides real-time guarantee and therefore wait-freedom if the number of cores and the non-atomic code executed allows the transactions to abort only a fixed number of times. For more details check equation 5 in section 6. We implement dynamically growing wait-free singly linked, doubly linked and limited capacity queues and evaluate their performance against their CAS and lock counterparts. We also carry out a worst-case execution time analysis of our queue implementations and bound their execution times proving them time-predictable and hence suitable for real-time systems. For our experiments, we use the Real Time Transactional Memory (RTTM) infrastructure available on JOP, an FPGA implementation of a multi-core Java processor system. To evaluate the scalability of our work, we ran our experimentation on the Azul machine using a large number of cores. The Azul machine has a runtime feature called Speculative Multi-address Atomicity (SMA) that attempts to run synchronized blocks transactionally. Our experiments show that TM based wait-free queue implementations perform better as contention increases and atomic sections grow in size, rendering TM as an ideal synchronization platform for wait-free algorithms.

The paper 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. The WCET analysis is presented in Section 6. Section 7 describes experimentation on the Azul machine and results. The paper is concluded in Section 8.

2. BACKGROUND

The Real-Time Specification of Java offers wait-free concurrent Queue classes as synchronization primitives for shared object access [28, 29]. Double Ended queues are preferred for real-time scheduling [26]. Traditional implementations of concurrent queues using locks have been proven to degrade performance and unsuitable for real-time systems as they cause blocking [23, 8]. As a solution, CAS-based non-blocking concurrent queues have been proposed and extensively analysed. But they do not always perform very well. Either because of the sheer number of CAS operations, of high failed CAS rates or due to overhead of maintaining consistent queue nodes in list based implementations [10, 26].

Work on lock-free queues started a couple of decades ago [27, 31]. The Michael and Scott first-in, first-out queue [12] 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 [10] by maintaining a doubly linked structure and reversing the direction of insert/remove operations. However, this comes at the cost of occasional an O(n) queue traversals to patch inconsistencies. Kogan and Petrank [9] proposed a wait-free implementation where higher priority threads help the lower priority peers to complete execution. However, their solution works only under certain system configurations. 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. Implementations resort to locks in such scenarios. We did not find literature on non-blocking implementations of dynamically growing bounded queues. Many other non-blocking algorithms and data-structures like concurrent hash tables and graph structures need MCAS [15, 16, 25]. Bounded wait-free queues are usually implemented using locks, queues are either optimized for reads or writes, a `WaitFreeReadQueue` has the write operations guarded with locks, while a WaitFreeWriteQueue has the read operations guarded with locks [29].

3. JOP

JOP, the Java Optimized Processor, is a hardware implementation of the Java Virtual Machine [17]. It is a time-predictable bytecode processor with real-time garbage collection capabilities. It is a RISC based stack computer that is capable of dynamically translating Java bytecodes into a stack based 'microcode'. It features a 4 stage pipeline each executed in a single processor cycle with the first stage performing the bytecode to microcode translation. It features a time-predictable instruction and data 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 Real Time Transactional Memory (RTTM) infrastructure, rendered JOP as the processor of choice for our experiments. We use the multi-core version of JOP [14] with four JOP cores for our experiments. Garbage collection for the multi-core version of JOP is under development, so we did not rely on GC for the queue implementations.

JOP is available for different targets like Altera and Xilinx FPGAs. The design flow [18] involves steps to generate, compile and program the JVM VHDL microcode onto the FPGAs; compile, download and execute Java applications on the programmed JVM. The Jopa, JOP Assembler assembles the microcoded JVM to generate the VHDL files which is compiled and programmed by Quartus via the ByteBlaster interface. The microcoded JVM is configured to load the Java application from the RS232 interface and start its execution. The Java application is first compiled using a Java compiler and then it is linked using the JOPizer tool. JOPizer links the class files generated and converts it to a JOP readable form using the Bytecode Engineering Library. The jop file is downloaded to the main memory of the Altera board using the RS232 interface.

3.1 Real-Time Transactional Memory on JOP

RTTM on JOP [20] is a time-predictable hardware implementation of transactional memory 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 cache local to each core that caches changed (write operations) data during a transaction. A set of tag memories (local to the core and non-cached) maintain the read locations. The processor state is saved before initializing a transaction. On a commit, the changed data in the local cache is copied atomically to the shared area using a global lock. A conflict is said to occur if the read set of one transaction interferes with the write set of another transaction. Intersection of write sets is not a conflict, as the writes are serialized during the commit.

Conflict detection happens only during a commit when all $n-1$ (on a n core multiprocessor) cores listen to the core that commits the transaction and not during local read/writes thus saving valuable CPU cycles. When a conflict is detected the transaction is aborted and restarted.

The real-time behaviour of such transactions is established by bounding the number of retries r to $n-1$ on a n core multiprocessor [19]. Assuming periodic threads, non-overlapping periods and execution deadline not exceeding the period, the WCET of any thread t is given by the equation

$$t_{wcet} = t_{na} + (r+1)t_{amax} \tag{1}$$

where t_{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. Since r is bounded, the WCET of any thread is bounded.

4. IMPLEMENTATION

We have implemented dynamically growing singly linked, doubly linked, and limited capacity wait-free queues using three different synchronization techniques, CAS, lock and transactional memory, resulting in ten variants of the queue. All the queues are FIFO queues with supported primitives 'insert at tail' and 'remove from head'. The different variants are explained below.

In our implementations, we use the Java synchronized keyword to simulate locks and the annotation @atomic for micro-transactions. The JOP hardware does not provide native support for CAS instructions. Hence, we simulate CAS using available synchronization infrastructure like locks and TM. This results in two different implementations for CAS, CAS_TM where a CAS call is a micro-transaction and CAS_LOCK where a CAS call is surrounded by a lock. In the paper, we use the term CAS as a generalization for both implementations. CAS_TM code snippet is as follows:

```
// CAS_TM
@atomic boolean CASHead(Node oldh, Node newh) {
  if (head == oldh) { head = newh;  return true;
   } else return false;
}
```

CAS_LOCK is similar except that `@atomic` is replaced with `synchronized`.

4.1 Wait-free Singly Linked Queue (SLQ)

The SLQ has been implemented in four variants using the 2 CAS, LOCK and the TM synchronization primitives. The LOCK and the TM variants are standard singly linked based FIFO queue implementations except that the 'insert' and the 'remove' methods are surrounded by either a LOCK or are micro-transactions. This implementation involves a special 'head' node to keep track of the queue-empty condition. Both the head and the tail pointers point to the 'head' node at the beginning and when the queue is empty. The following code snippet summarizes the implementation.

```
final static class Node {
  final Object value;
  volatile Node next = null;
}

volatile Node head = new Node(null);
volatile Node tail = head;

void insert(Node n) {
        tail.next = n;
        tail = n;
}

Object remove() {
   if (head.next != null) {
     head = head.next;
     return head.value;
   }
   return null;
}
```

In the remove method, the removed node is retained as the special 'head' node until the next node is removed. This does not affect the number of retries.

The CAS based SLQ variants implement the non-blocking concurrent queue algorithm described in [12]. Our implementation closely follows the Java implementation of the Java utility class 'java.util.concurrent.ConcurrentLinkedQueue'. [1] It uses primitives CASHEAD, CASTAIL and CASNEXT to modify the head, tail and the node pointers respectively. The multi-core version of JOP is yet to introduce garbage collection. We make sure that the node insertions and removals follow the intended order through verification.

4.2 Wait-free Doubly Linked Queue (DLQ)

The LOCK and TM based implementation of the DLQ is similar to the SLQ implementation except that the nodes have both 'next' and the 'previous' pointers and the 'insert' and the 'remove' primitives modify both these node pointers. As in SLQ, the queue primitives are surrounded by either a '`synchronized`' keyword or annotated with '`@atomic`'.

The CAS variants of the DLQ implement the algorithm described in [10]. The algorithm uses an approach where the doubly linked node pointers are updated using regular 'store's resulting in the usage of only one CAS instruction each to insert and remove queue nodes. It is optimistic in the sense that the algorithm assumes that queue inconsistencies will not occur. But, if they do, the inconsistencies are fixed in subsequent operations which can be quite expensive. This algorithm has been proven to perform better than many wait-free linked list based FIFO queues but it suffers with a problem where the queue links are consistent only in one direction and needs to be corrected when necessary. Our implementation, as explained in the previous section, do not use pointer tags or counters.

4.3 Wait-free Limited Capacity Queue (LIMQ)

The LIM is a doubly linked queue with capacity constraints. This requires that the queue size is maintained and is checked against the capacity during insertions for queue-full condition. Also, the insertion and the removal primitives now have to atomically increment and decrement the queue size. This problem is usually solved by using a Multi-Word CAS primitive or by limiting the queue to a single reader or a single writer [29]. Hence we do not present implementations for the CAS variants. We only implement the LOCK and the TM variants. The following code snippet summarizes the implementation.

```
final static class Node {
        final Object value;
        volatile Node next = null;
        volatile Node previous = null;
}

volatile Node head = new Node(null);
volatile Node tail = new Node(null);
final int capacity;
volatile int size = 0;
```

[1]http://fuseyism.com/classpath/doc/java/util/concurrent/

```
boolean put_tr(Node n){
        boolean done = false;
        if (this.size < this.capacity) {
                n.previous = tail.previous;
                n.next = tail;
                tail.previous.next = n;
                tail.previous = n;
                done = true;
                this.size++;
        }
        return done;
}

Object get_tr(){
        Node n = null;
        Object value = null;
        if (head.next != tail) {
                n = head.next;
                head.next = n.next;
                n.next.previous = head;
                this.size--;
                value = n.value;
        }
        return value;
}
```

As in SLQ and DLQ, the queue insertion and removal primitives are associated with one of 'synchronized' or '@atomic'.

5. EXPERIMENTATION AND EVALUATION

The experimentation environment is an FPGA programmed with a symmetric shared-memory multi-processor hardware system with four JOP cores. As hardware platform we us an Altera DE2-70 Development board[2] consisting of a Cyclone II EP2C70 FPGA. The Altera board contains 64 MB SDRAM, 2 MB SSRAM and an 8 MB Flash Memory and I/O interfaces such as USB 2.0, RS232, and a ByteBlasterMV port. Each JOP core has a core local 4 KB instruction cache and 1 KB stack cache. The Cyclone FPGA was programmed to simulate the afore-mentioned symmetric shared-memory multi-processor environment.

To evaluate and compare the various synchronization primitives for concurrent wait-free queues, we conducted experiments using a producer-consumer framework on a four-core symmetric multi-core system each capable of executing Java bytecode. Each of the four cores executed an independent shared memory thread with one of producer, consumer or a consumer-producer combination functionality embedded in them. The queue nodes were exchanged among the concurrent threads rendering the queue head, tail pointers and nodes as collision points. Synchronization was achieved through CAS, LOCK and TM primitives. Data on the execution time and other TM properties were collected and compared. The rest of the section explain the framework and the evaluation in detail.

5.1 Producer-Consumer Framework

The experiments were conducted using a Producer-Consumer Framework where producers produce nodes and insert them at the tail of the queue while the consumers remove the nodes from the queue head and consume them. Each experiment involved two wait-free queues A and B, both empty at the beginning of the experiment. The producer and the consumer functionality were performed by independent JOP threads referred to as Inserter and Remover Threads each of which executing on a separate JOP core. The Inserter, in a loop, inserts a specified number *num* of queue nodes into A and similarly, the Remover removed *num* number of nodes from B. Another thread called the Mover Thread removes a node from A and inserts the same node into B. Each experiment had two instances of the Mover thread each running on a separate JOP core. Each Mover processes exactly $num/2$ queue nodes.

The queue insertions and removal operations were performed atomically using the four synchronization variants mentioned above. Note that the Java programs associated with the threads did not incorporate any synchronization primitives, meaning each iteration of the various threads executed without any knowledge of the status of other threads. Only the queue insertion and removal methods guaranteed atomicity. Such a system of threads, composed of atomic and a non-atomic sections, confirms with the thread model used to establish real-time bounds on the number of retries in [19].

In each experiment, all the four threads were started simultaneously. The collision points in such an experimental setup are the head and tail pointers of the queues A and B and the pointers associated with the queue nodes exchanged among various threads. The presence of two Mover Threads increases the contention as two threads may try to remove the same node from queue A or try to insert different queue nodes at the same time.

5.2 Experimentation

Experiments were conducted to record the execution time, number of commits, retries, size of read and write sets of the different queue implementations. The combination of singly/doubly linked, limited capacity queues and the four synchronization primitives resulted in 10 different queue implementations, four variants of the singly linked queue, four of the doubly linked and two variants of the limited capacity queue as it is not feasible to implement the CAS_TM and CAS_LOCK variants. For each such variant of the queue, the experiment involved executing the Inserter, Mover and Remover threads to completion for a specified value of *num*. The value of *num* was varied from 10 to 5000 and for each value, the above mentioned parameters were recorded.

5.3 Evaluation

For the comparison and evaluation of various synchronization techniques, we are interested in the total number of commits, retries and the maximum size read/write sets of an experiment (involving all threads/cores) than those of individual cores. The effect of workload and transactional data size on the number of commits/retires and the read/write sets of an individual core has been dealt in detail by [19].

5.3.1 Transactional Read-Write Sets

Table 1 shows the sizes of the read, write and the union of the read and the write sets for the TM and the CAS_TM cases. The sizes are 0 for the LOCK and the CAS_LOCK cases as they don't use any transactions. It can be noted that there is a significant increase in the sizes of the read/write

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

		Read Set	Write Set	Read-Write Set
TM	SLQ	7	2	7
	DLQ	11	4	12
	LIMQ	11	5	12
CAS_TM	SLQ	3	1	3
	DLQ	3	1	3

Table 1: Read, Write, and Read-Write sets of TM and CAS_TM based queues

sets from SLQ to DLQ and LIMQ for the TM case. This can be attributed to the extra pointers that need to be modified while inserting and removing nodes from a DLQ as compared to a SLQ. Limited capacity queue (LIMQ) is implemented as a doubly linked list with a 'capacity' parameter. Hence there is a small increase in the write set size. However, the read and write set sizes for the CAS_TM based queues do not vary. Although there is an increase in the number of queue node pointers from SLQ to DLQ, the DLQ algorithm described in [10] uses one CAS operation per queue operation which is comparable to the SLQ case. Also, a comparison of the read/write set sizes in the TM case with those of CAS_TM reveals that the sets are smaller in the case of CAS_TM. This is because, CAS implementations of wait-free queues involve smaller transactions as compared to the TM case. For example, inserting a node in a CAS based implementation of SLQ requires two CAS operations, one to modify the tail's next pointer ($CASNEXT$ operation) and the other to modify the tail ($CASTAIL$ operation) itself involving two transactions as compared to a single transaction in the TM case where both the pointers are modified in one attempt.

5.3.2 Transactional Commits

In the queue versions CAS_TM and TM, the number of transactional commits vary depending on the number of CAS operations and transactions used. Table 2 indicates the number of commits per queue operation for the various queue implementations. Note that the commits stated here does not include the commits associated with a retried transaction.

		Insertions	Removals
SLQ	CAS_TM	2	1
	TM	1	1
DLQ	CAS_TM	1	1
	TM	1	1
LIMQ	TM	1	1

Table 2: Commits per queue operation

Table 2 indicates that all the TM implementations involve only one commit per queue operation. Because, no matter how may queue node pointers and/or other queue parameters are to be updated, TM based implementations can atomically update in one transaction. However, it is not the case with CAS_TM based implementations. In the case of CAS_TM, each CAS operation corresponds to a commit. Note that the number of commits recorded by SLQ is higher than the case of CAS_TM based DLQ. This is because our CAS based DLQ uses only one CAS per queue operation, while SLQ uses two for insertion.

A few general points worth noting about commits are as follows. The CAS based implementation records more commits than the TM versions because CAS implementations involve more transactions than the corresponding TM implementations. An example is CAS_TM based SLQ. As the number of queue node pointers increase, number of CAS instruction increases and hence the number of commits. For example, the double ended queue implementation described in [26] requires several CAS instructions to insert/remove nodes from a doubly linked list. In the case of TM, increase in queue node pointers may not increase commits as all the necessary pointer modifications can be atomically carried out using a single transaction. Also, as the number of processed nodes increase, the number of commits increase. As the number of transactions increase, there is a high probability that the number of retries also increases as transactions being committed may conflict with other transactions.

5.3.3 Singly Linked Queues

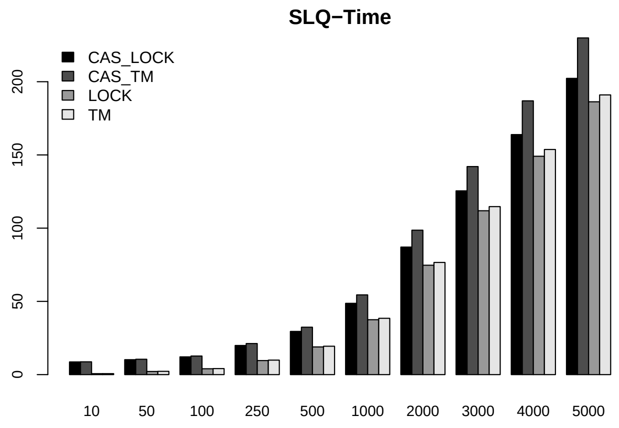

Figure 1: SLQ Time The x-axis gives number of nodes and the y-axis gives the execution time in milli-seconds.

Figure 1 plots the execution time to insert, move and remove a specified number of nodes from the singly linked queues. The x-axis indicates the number of nodes used in the experiment, we chose a sample of the small number of nodes followed by a linear increment starting from 1000. The bars indicate the time taken to complete the experiment when different synchronization methods are used. Time is measured from the instance when the insertion of the first node is started till the removal of the last node is completed. It can be noted that, as the number of nodes processed increase the average execution time increases almost linearly due to an increase in the number of locks, transactions and retries.

Note that the CAS based implementations are much slower than their LOCK and TM based counterparts. This is because CAS based wait-free queue implementations involve two CAS operations (resulting in two transactions/operation) in the best case as opposed to one in the case of TM and LOCK. CAS based algorithms, unlike their TM and LOCK counterparts, also run additional checks to maintain queue consistency adding to execution time. Although execution times in the case of TM is 17% lower compared to the CAS cases, it is 2.5% higher than the LOCK case. The higher execution times of the TM implementation relative to that of

LOCK can be attributed to the small sizes of the read/write sets and shorter atomic sections in singly linked queues. For example, in a singly linked list, an insert operation involves the modification only of a pointer and the queue tail. As a result, locks are held for a short period reducing overall waiting time. However, in the case of TM, retries, conflict detection and other transactional memory overhead is high as compared to the time lost in waiting for locks.

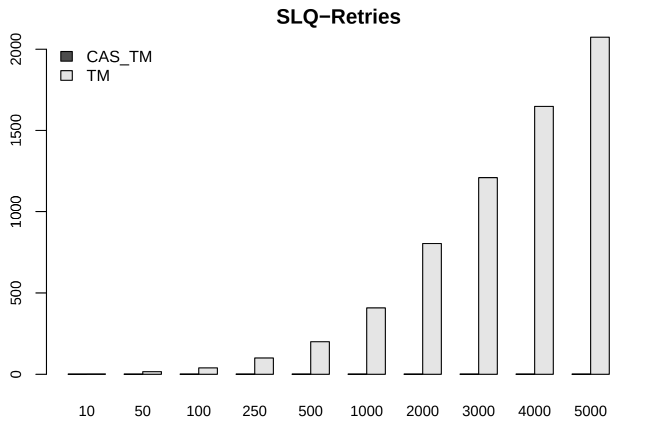

Figure 2: SLQ Retries The x-axis gives number of nodes and the y-axis gives the total number of retries.

Figure 2 plots the number of retries in the system. Note that retries are plotted only for the CAS_TM and TM cases. LOCK based implementations do not involve retries. Figure 2 indicates that CAS_TM records very low number of retries (0 or 1) compared TM. This is because, CAS implementation do not modify queue/node pointers if they are already modified by other threads in the system. Also, CAS_TM uses transactions of a size of a word. Since the transactions are small and quick in nature, the number of conflicting transactions reduce bringing down the retry count.

5.3.4 *Doubly Linked Queues*

Figure 3 plots the execution time for doubly linked queues.

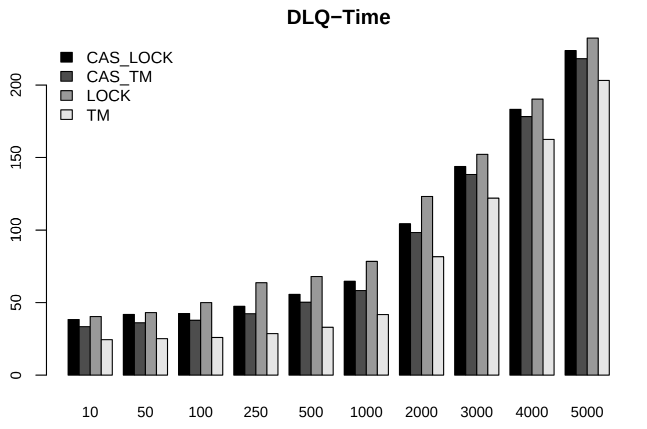

Figure 3: DLQ Time The x-axis gives number of nodes and the y-axis gives the execution time in milli-seconds.

Figure 3 indicates that CAS variants are slower compared to the TM variant by 8%. This is because CAS implementations of doubly linked queues add significant overhead in maintaining consistent queue node connections as insertions and removals involve multiple pointer changes. Such implementations involve additional checks to maintain queue structure consistency and also allow non fatal inconsistencies to occur and correct them during subsequent queue operations [10] [26]. It is interesting to note that our CAS_TM based DLQ implementation performs better than its SLQ counterpart. This is because the DLQ, as explained above, replaces costly CAS operations with regular stores using only one CAS instruction each to insert and remove nodes as opposed to two for inserting nodes into SLQ. The number of commits of the CAS version of DLQ is smaller than that of the SLQ. This performance gain is also supported by the experiments and results discussed in [10]. Figure 3 also indicates an 15% performance gain in the case of TM compared to LOCK. This gain can be completely attributed to bigger read/write sets. Locks are held for longer periods increasing waiting times hence increasing the execution time in the case of LOCK. This illustrates the motivation for using Transactional Memory based micro-transactions for shared data synchronization. Another advantage of TM based DLQ over the CAS based ones is in the fact that the TM implementation maintains the queue node pointers consistent at all times without extra effort to correct inconsistencies. Hence it is straight-forward to scale the TM implementation to construct a Double Ended Queue (Deque). Sundell and Tsigas [26] discuss lock-free dequeues using single-word CAS and Fetch & Add (FAA) operations. But their algorithm, like others, involves multiple CAS operations, consistency checks and correcting inconsistencies, which can be avoided with TM based Deque implementations. The number of retries in the DLQ experiment follows the same trend as SLQ. As in SLQ, the number of retry in the case of CAS_TM is negligible.

5.3.5 *Limited-capacity Queues*

In our implementation, limited capacity queues are doubly linked queues with limited capacity. Such queues increase contention especially among Mover threads 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 limited capacity queues were only conducted using the LOCK and the TM variants as it is traditionally not possible to have single word CAS based bounded queue implementations.

Figure 4 compares the execution time of the LOCK based limited capacity queue with that of the TM based queue. The plot indicates a 19% increase in the performance of the TM based queue. Locks are held longer by threads due to the additional check and increment operations while insertion and a decrement operation during removal. This significantly increases the execution time. Note that the TM implementations do not add additional overhead for the queue-full check as the queue 'capacity' is a read-only queue parameter. The retries and commits follow the same trend as the singly and doubly linked queues.

5.3.6 *Summary*

The singly linked, doubly linked and the limited capacity queues test the transactional memory infrastructure and

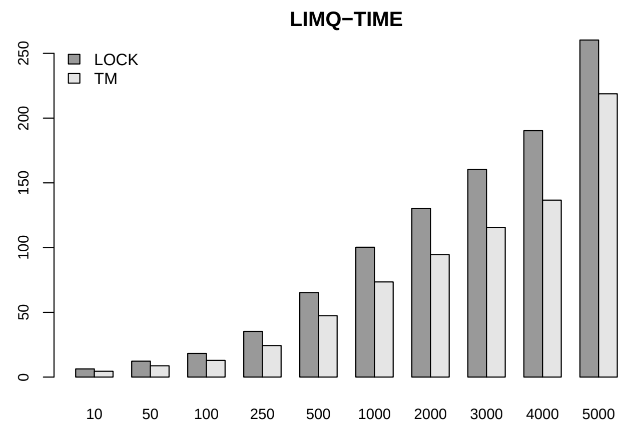

Figure 4: LIMQ Time The x-axis gives number of nodes and the y-axis gives the execution time in milli-seconds.

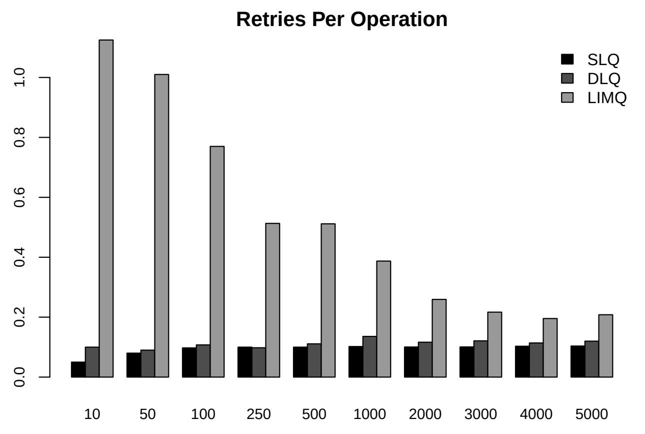

Figure 5: Retries Per Operation The x-axis gives number of nodes and the y-axis gives the number of retries per queue operation.

compares it with other synchronization methods with increasing degree of contention and increasing read/write set sizes. Table 1 shows the increase in the read/write set sizes for the three queue types. Figure 5 indicates the average number of retries per queue operation for SLQ, SLQ and LIMQ. The average is calculated by dividing the total retries by the total number of queue operations (4 operations per queue node processed). Note that the retries of DLQ are higher by a small amount compared to that of SLQ. This is because DLQ has bigger read/write sets. But retries per operation is two folds higher in the case of LIMQ compared to DLQ and SLQ. LIMQ records high retries per operation when the nodes processed are less. We discount these cases because as the number of processed nodes increases, retries per operation stabilizes. The high value in the case of LIMQ compared to SLQ and DLQ is because in LIMQ experiments, the queue size is modified both in insertion and removal routines increasing contention among threads. When the contention and the read/write set sizes are low, as in the case of singly linked queues, LOCK outperforms TM. But when contention and read/write set sizes increase as in the case of doubly linked and limited capacity queues, transactional memory based implementations perform better. Note that TM based implementations make dynamically growing bounded buffer wait-free queues feasible which otherwise need multi-word CAS instructions. Figure 4 shows that LOCK based bounded buffer queues are quite expensive. In our experiments, TM based implementation performs better than LOCK by 19%.

5.4 Other Experiments

We conducted many different experiments on the various implementation of the queues. Two of which 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 significant change in these parameters with the change in queue capacity. We noticed a slight increase in the execution time when the capacity was lowered to very small values. This is due to the threads finding that the queue is full.

2. We also conducted experiments by changing the number of Mover, Inserter and Remover threads but we did not notice a significant change in the behaviour, performance or other parameters. No new trends were observed.

6. WCET ANALYSIS

In this section, we analyze the worst-case execution time (WCET) properties of the threads that use our micro-transactions based queue primitives. For the WCET analysis we use the WCA tool [21], which is part of the JOP distribution. We carry out the WCET analysis for the LIMQ since it consists of the largest atomic sections and LIMQ is presented as a contribution of this paper. LIMQ is particularly interesting because both insert and remove primitives access the queue size and hence the contention is high compared to unbounded queues.

To prove any system of threads to be real-time, one needs to be able to bound the execution times of all the threads. The property that all the threads using our queue primitives complete their execution is also sufficient to prove that our queue implementation results in a wait-free system. The set of arguments stated in the analysis is similar to the ones used to prove the real-time property of the micro-transactions themselves (Section 3.1), explained in detail in [19]. The following explains the assumptions and analysis.

A major concern when calculating the WCET of a thread using transactional memory is bounding the number of retries of its transactional scopes. A thread is composed of atomic and non-atomic sections. For our analysis we assume the that the only atomic sections that a thread executes are of our queue primitives and the non-atomic sections can be executed independent of other threads without contention. In this model, a thread finishes one iteration of its execution if all of its atomic sections finish execution, i.e., if all the transactions successfully commit. This is straightforward when there are no retries, the transactions commit in the first attempt. However, if there are retries, it is possible that a low priority thread never completes because other higher priority threads keep entering atomic sections and successfully commit. This problem can be solved if we divide time into periods of equal length and limit the number of transactions that can execute in a given period. In the thread model described in [19], if a thread successfully commits its transaction, it does not participate again. Every successful commit reduces the number of contending threads and finally the last thread commits and completes the transaction. This ensures that even the lowest priority thread that has entered a transaction completes. An example is described as follow:

Consider a system of n threads and let t_{amax} be the time taken by the largest atomic section, 'insertion' primitive in our LIMQ implementation. If we define the period p to be $n \times t_{amax}$ time units and specify that no more than n transactions execute in that period, then the number of retries is limited to $n-1$ and all threads finish execution. This can be easily achieved by specifying that threads spend at least

$$p = n \times t_{amax} \quad (2)$$

time units between any two transactions. This assumption is technically the same as Lemma 1 of [19] and period p corresponds to the resolution time t_r (Equation 10 of [19]).

The implementation of the LIMQ insertion (`put`) and removal `get` routines are composed of atomic and non-atomic sections. For example, in the insert routine, memory allocation for the queue node is a non-atomic part. These routines invoke the afore-mentioned `put_tr` and `get_tr` which are the actual transactional part that access queue parameters. The `put` routine creates a new queue node, and loops on `put_tr` until the object is added to the queue. This is independent of a transaction roll-back, `put_tr` is called more than once only if the queue is full. Similarly, `get` loops on `get_tr` if the queue is empty until an object is retrieved. For our analysis, we assume that the queue has enough capacity and the queue is not empty. Note that the looping can be avoided by having the `put` and `get` routines return an error.

We calculated the cost of the atomic sections as well as the overall cost of the queue operations. Table 3 shows the number of cycles for the atomic sections and the queue routines when the transaction succeed in the first attempt. To calculate these numbers, we used the cycle accurate WCET analysis as described in [21]. Cycles required for `put` is significantly larger than `put_tr` unlike `get` and `get_tr` because `put` includes `Node` creation that takes 347 cycles. These numbers does not include a small bounded transactional overhead. Note that the cycle values are indicative in nature. Number of cycles can be reduced by inlinig the atomic sections `put_tr` and `get_tr` methods into their callers removing the cost of the method invocation.

put_tr	put	get_tr	get
315	807	306	475

Table 3: Number of cycles required for the different methods within LIMQ (assuming no retries)

Note that out of 807 cycles spent in the `put` routine, 315 cycles correspond to `put_tr` which are executed in transactional scope and are vulnerable for retries. Similarly, out of total 475 cycles of `get`, 306 cycles that correspond to `get_tr` are vulnerable for retries. Using these values in equation 2, the period p will have a value ($n \times 315$), as $t_{amax} = 315$ cycles are required by the largest atomic section `put_tr`.

Let t_{naimax} be the maximum of the execution times of the non-atomic sections of `put` and `get` routines and t_{nae} be the execution time of a thread t outside of `put` or `get`. Note that t_{naimax} can be calculated by

$$t_{naimax} = t_{put} - t_{amax} \quad (3)$$

where t_{put} is the time required for `put` to execute without any retries. We use t_{put} as its non-atomic section takes the maximum number of cycles to execute. The non-atomic section execution time t_{na} of equation 1 is given by

$$t_{na} = t_{naimax} + t_{nae} \quad (4)$$

Revisiting equation 1,

$$t_{wcet} = t_{nae} + (m)(t_{naimax} + (n \times t_{amax})) \quad (5)$$

where m is the number of atomic sections in a thread each separated by $(n \times t_{amax})$ time units.

7. AZUL

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 Speculative Multi-address Atomicity (SMA). With this flag set, the Azul runtime attempts to run the synchronized parts (java `synchronized` keyword) transactionally using hardware transactional support. Azul is known to perform well for small transactions. Unlike experimenting with JOP, since 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 algoritm scales up to 64 cores. We experimented only with the LOCK and the CAS_LOCK versions both with and without SMA support. In the rest of the text, NoSMA indicates that the SMA property is disabled. With SMA flag enabled CAS_LOCK corresponds to CAS_TM and LOCK to TM. Also, the number of each node processed should be a multiple of the number of threads in Azul. So, we carry-out experiments starting from 4K (K = 1024) nodes to 1024K nodes. The Azul machine is an Azul Vega 3 3310B, with two 54-core processors and 48GB of RAM. The benchmark ran on top of the Azul Virtual Machine with the Concurrent Pauseless GC [3].

Given that the synchronized code scopes are small across 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. The results displayed are using 16 Inserter, 32 Mover and 16 Remover threads (a total of 64 threads). The number of threads have been chosen to use as many cores from the Azul machine (108 cores available) without overflowing. The figures below are based on the average of 3 runs with thread configuration described above. Experiments were also conducted starting from 4 up to 128 total threads with similar results.

Figure 6 shows the execution time of SLQ on the Azul machine. For most node sizes, the CAS_LOCK version without SMA support is the slowest followed by the CAS_LOCK with SMA support. This is because NoSMA acquire blocking locks and SMA use transactions. In the Azul experiments, the number of threads are high compared to JOP. This increases the contention and hence the number of failed

CAS instructions. This is one of the main reasons for low CAS performance over non CAS versions. The execution time of LOCK with SMA performs slightly better than the NoSMA case. But in a few experiments NoSMA performed better than the SMA experiment. Although it is difficult to state the exact reason why NoSMA case wins over SMA without knowing SMA's implementation, a possible reason could be that the atomic sections are very small and lock contention is very low while SMA enabled version is paying for the transactional check overhead.

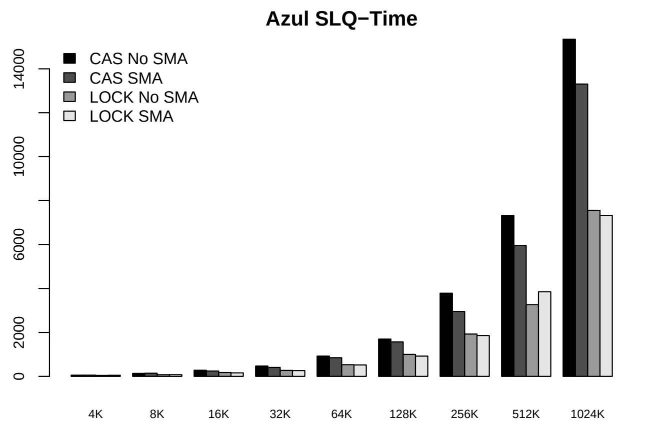

Figure 6: Azul SLQ Time The x-axis gives number of nodes and the y-axis gives the execution time in milli-seconds.

Figure 7 shows the results of the DLQ experiment on Azul machine. Similar to the SLQ results, the CAS versions are slower than the LOCK (with and without SMA) versions. Although we do not have data on the number of times the `fixlist` routine [10] was invoked, we believe high contention increases queue inconsistencies and hence calls to `fixlist`. As a result, CAS based versions are much slower on Azul as compared to JOP. It can be noted that the LOCK version with SMA enabled performs better than its NoSMA counterpart. The analysis is similar to that of JOP. As the number of pointers increase, atomic sections grow increasing locking periods. It can be noted that under high contention, SMA case performs better than NoSMA.

Finally, Figure 8 shows the results of the LIMQ experiments. As stated in the previous sections, LIMQ offers the largest read/write sizes and high contention as compared to other queue implementations. In such a scenario, LOCK with NoSMA experiments not only acquire locks and hold for longer time but also contend for the lock during both insertions and removals. Similar to the evaluation on JOP, SMA case performs better as locking increases the waiting time and hence overall execution time. SMA enabled experiments perform 9% better than the NoSMA case.

8. CONCLUSION AND FUTURE WORK

This work considers Transactional Memory (TM) as an alternative to CAS and LOCK based synchronization primitives. We showed that concurrent algorithms requiring multi-word CAS primitives can be implemented using TM in a straight-forward way. Our experiments suggest that TM based implementations of concurrent non-blocking queue algorithms perform better than the CAS based implementations. We also showed that as atomic sections and contention grow, TM based implementations perform better. The experiments on Azul platform indicate that TM based implementations are scalable. We have analysed the worst case execution times of a system using our queue implementations and proved that our TM based queue implementations result in a wait-free system and are suitable for real-time applications.

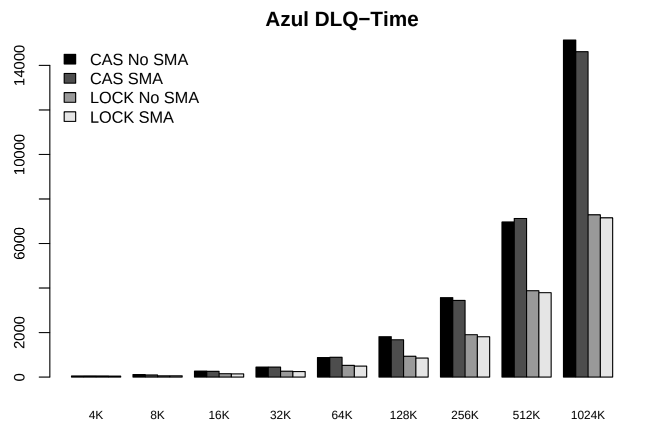

Figure 7: Azul DLQ Time The x-axis gives number of nodes and the y-axis gives the execution time in milli-seconds.

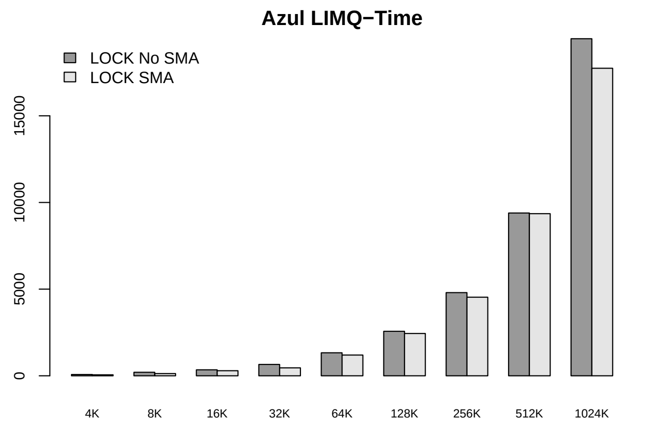

Figure 8: Azul LIMQ Time The x-axis gives number of nodes and the y-axis gives the execution time in milli-seconds.

Future work involves exploring larger data-structures like double ended queue, graph structures and hash tables. Analysing TM performance on larger systems with increased contention is useful. The wcet analysis states that in a given period only a fixed number of transactions are to be executed. Such a requirement raises questions like which transactions to execute and based on what parameters (thread priority, deadline etc.) such a decision is to be made. Such questions can be explored from a scheduling theory perspective.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0958465, 0811691 and

0720652.

9. REFERENCES

[1] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 316–327, 2005.

[2] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[3] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *International Conference on Virtual Execution Environments (VEE)*, pages 46–56, 2005.

[4] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News*, 32:102–, 2004.

[5] T. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, pages 265–279, Oct 2002.

[6] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

[8] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 1993 International Symposium on Computer Architecture*, 1993.

[9] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP)*, pages 223–234. ACM, 2011.

[10] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free fifo queues. *Distributed Computing*, 20(5):323–341, 2008.

[11] V. J. Marathe and M. L. Scott. A qualitative survey of modern software transactional memory systems. Technical report, 2004.

[12] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[13] F. Nemati, J. Kraft, and T. Nolte. Towards migrating legacy real-time systems to multi-core platforms. In *Proceedings of the Work-In-Progress (WIP) session of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08)*, pages 717–720, September.

[14] C. Pitter and M. Schoeberl. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst.*, 10(1):9:1–34, 2010.

[15] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. Technical Report UCAM-CL-TR-639, University of Cambridge, Computer Laboratory, Sept. 2005.

[16] S. Ramamurthy. *A Lock-Free Approach to Object Sharing in Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill, 1997.

[17] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[18] M. Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Number ISBN 978-1438239699. CreateSpace, August 2009.

[19] M. Schoeberl, F. Brandner, and J. Vitek. Rttm: real-time transactional memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 326–333, 2010.

[20] M. Schoeberl and P. Hilber. Design and implementation of real-time transactional memory. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL)*, pages 379–284, 2010.

[21] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[22] N. Shavit and D. Touitou. Software transactional memory. In *ACM symposium on Principles of distributed computing (PODC)*, pages 204–213, 1995.

[23] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts (7. ed.)*. Wiley, 2005.

[24] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC)*, pages 338–339, 2007.

[25] H. Sundell. Wait-free multi-word compare-and-swap using greedy helping and grabbing. In *Parallel and Distributed Processing Techniques and Applications*, pages 494–500, 2009.

[26] H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *J. Parallel Distrib. Comput.*, 68(7):1008–1020, 2008.

[27] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, April 1986.

[28] P. Tsigas and Y. Zhang. Efficient wait-free queue algorithms for real-time synchronization. Technical report, Department of Computing Science, Chalmers University of Technology, 2002.

[29] P. Tsigas, Y. Zhang, D. Cederman, and T. Dellsen. Wait-free queue algorithms for the real-time java specification. In *IEEE Real Time Technology and Applications Symposium*, pages 373–383. IEEE Computer Society, 2006.

[30] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30:66–75, 2010.

[31] J. D. Valois. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.