

Survey of Concurrent Priority Queue Algorithms*

Kristijan Dragičević, Daniel Bauer,
IBM Research
Zurich Research Laboratory
{kdr, dnb}@zurich.ibm.com

Abstract

Algorithms for concurrent data structures have gained attention in recent years as multi-core processors have become ubiquitous. Using the example of a concurrent priority queue, this paper investigates different synchronization methods and concurrent algorithms. It covers traditional lock-based approaches, non-blocking algorithms as well as a method based on software transactional memory. Besides discussing correctness criteria for the various approaches, we also present performance results for all algorithms for various scenarios. Somewhat surprisingly, we find that a simple lock-based approach performs reasonable well, even though it does not scale with the number of threads. Better scalability is achieved by non-blocking approaches.

1 Introduction

In the past, CPU performance mainly increased as a function of the operating clock frequency. Physical limits have caused a shift in processor design. Today, ever more processing cores are placed on a single chip. Application software exploits multi-core designs by embracing parallel execution. Consequently, many sequential data structures have to be adapted in order to scale well on multi-core systems. This is especially challenging for concurrently accessed and manipulated data structures because mechanisms to avoid inconsistencies are needed, which incur additional overhead. The prevalent way of handling concurrency is to use locking as a synchronization mechanism between threads. It is well-known that coarse-grained locking approaches lack scalability for increasing numbers of threads as they prevent parallel execution. On the other hand, solutions such as fine-grained locking that use multiple locks often prove difficult and error-prone in both design and implementation. Lock-free approaches try to overcome the disadvantages of lock-based methods. They typi-

cally implement concurrency control using atomic instructions such as compare-and-swap.

This paper discusses the differences between these methods and compares the scalability and performance of representative implementations for each approach. We do this by using the priority queue data structure as the case study. Priority queues are fundamental data structures often used as basic components in more complex systems. As the highest priority element in the queue represents a contention point, it is notoriously difficult to implement it efficiently for concurrent applications. Our motivation to study this is to find the most practical solution for the messaging system *Tempo* [3]. *Tempo* is a lightweight publish/subscribe messaging system that uses a priority queue as the basic data structure for its message scheduler. As *Tempo* is implemented in Java, we also use Java as the runtime environment for the algorithms presented here.

The paper is structured as follows. Section 2 gives an overview of the priority queue algorithms investigated including the correctness conditions that are fulfilled by each of them. Section 3 describes the performance test scenarios and discusses the results. Section 4 discusses related work. Finally, Section 5 provides a summary and conclusion. [4] is an extended version of this paper.

2 Concurrent Priority Queue Algorithms

A priority queue is an abstract data type with two operations. The *put*(x) operation adds the element x with priority $x.p$ into the queue, where $x.p$ is an integer value from a given priority range. The *get*() operation retrieves the element with the highest priority from the queue, provided the queue is not empty. This paper studies several concurrent priority queue algorithms that are classified into lock-based approaches (using a mutual exclusion lock), non-blocking algorithms (building on atomic operations such as compare-and-set (CAS)), and approaches based on software transactional memory (STM, [15]).

Even though each algorithm presented here implements a priority queue, the correctness condition that each ap-

*IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008. 14-18 April 2008

proach fulfills is different. A correctness condition defines how an algorithm behaves during parallel execution. The strongest condition is *linearizability* [9], also known as strict or atomic consistency. An algorithm fulfilling the *linearizability* correctness condition matches closest what intuitively would be considered as the parallel pendant of its sequential counterpart. A weaker condition is Lamport’s sequential consistency [12]. This condition is generally used as a correctness condition for transactions in databases and distributed systems.

Based on the properties introduced, the algorithms can be characterized as follows:

2.1 Lock-based Approaches

The classic approach to concurrency control is to use mutual-exclusion locks. We investigate three algorithms in this category.

Coarse-lock The coarse-lock approach uses a single global lock that protects access to a binary heap. Therefore, it prohibits parallelism altogether. The global lock forces *put* and *get* operations to be executed strictly sequentially and thus provides strict consistency.

Hunt heap A refinement of coarse-lock was presented by Hunt et al. [11]. It uses mutual exclusion locks to protect the heap size variable as well as each node in the binary heap. Furthermore, the “bit-reversal” technique is applied, which is basically the same idea as the LR-algorithm described in [1]. Consequently, Hunt’s fine-grained locking approach increases parallelism. This approach is linearizable but suffers one noticeable disadvantage. It may end up in a near-deadlock situation that occurs when the capacity of the queue is reached.

Parallel Fibonacci heap This algorithm, described in [10], uses a set of Fibonacci heaps that are synchronized using locks. The heaps are independent of each other and contain each a distinct subset of the elements. The performance of the algorithm depends on the number of heaps. The quality of the removed nodes depends on various parameters like the size of the “promising list” and the so-called “strictness parameter”, see discussion in [10]. This evaluation considers an algorithm with $\frac{2}{3}(n+1)$ heaps and the same number of promising elements, where n is the number of threads. This algorithm is neither linearizable nor sequentially consistent. To increase parallelism, it uses a randomization technique that spreads out operations to different heaps. Parallelism and thus performance increases with the number of heaps. At the same time, randomness also increases. The worst-case behavior results in random results, and it is arguable whether this algorithm qualifies as a valid concurrent priority queue implementation.

2.2 Non-Blocking Algorithms

The second class of algorithms is known as non-blocking algorithms. They refrain from using locks and instead are based on instructions such as compare-and-swap as the basic means of concurrency control. Non-blocking algorithms are further classified as either *wait-free*, *lock-free*, or *obstruction-free* [7]. This paper investigates two non-blocking algorithms.

Lock-free skip-list Sundell and Tsigas present in [17] a fast lock-free priority queue that is based on a sorted skip-list. It applies a helping strategy that is essential to achieve the lock-free property as it allows a task to continue even though another task has unfinished work. Sundell’s algorithm is lock-free and linearizable. The version of the algorithm described in [17] is not quite as general as other implementations because it only allows a single element per priority. However, there exists an accelerated, commercial version of this algorithm that has been modified to deal with this issue. For our performance tests, the version published in [17] is used.

Quantizing queue This algorithm has been developed for the Tempo messaging system [3]. This method is similar to the simple bounded range priority queue algorithm described in [16]. It operates on a bounded priority range that quantizes priorities into a fixed number of priority levels. Elements having the same priority level are stored in a lock-free FIFO queue described by Michael and Scott in [13]. The algorithm is non-blocking because the underlying FIFO queue is lock-free and thus non-blocking. It is sequentially consistent, but not linearizable, in contrast to a statement in [16]. The proof for non-linearizability is given in [4]. Since the algorithm provides a fixed number of priorities levels within a given priority range, it is less general than other algorithms.

2.3 STM-Based Algorithm

STM [15] is a relatively new programming paradigm that has recently been an area of intense research. The basic idea is to declare a piece of code as being an atomic block if its effects must appear atomic. The execution of an atomic block is called a transaction. If a conflict with other transactions occurs, the computation is discarded and the block is reexecuted from the beginning. STM implementations can be lock-based [5] or non-blocking [6]. We use the DSTM2 factory [8], which is non-blocking and, with the aid of a contention manager [14], practically lock-free. This paper considers a single priority queue that is based on STM. It implements optimizations to the naive approach that merely declares the entire queue access operations as being atomic.

STM This algorithm uses a binary heap where *put* and *get* operations are executed as a sequence of transactions. It has similarities with the *Hunt heap* concerning the insertion policy of elements. As a novelty, it introduces a new value with which nodes can be tagged to be a *hole*. The performance of this algorithm mainly depends on the underlying STM implementation. Its properties are also inherited from the STM used. We reimplemented an existing C-solution of an STM-based binary heap by using the DSTM2 [8] library. As the DSTM2 factory is designed for functional and proof-of-concept experiments rather than for performance, the test results are not representative for this implementation. The C-implementation using Fraser’s solution [6] shows a much better performance. It also includes optimizations that are not implemented in the Java-version.

The description of the six techniques presented above shows that they are not simply comparable in terms of raw performance numbers. Independently of the results of our tests, the characteristics explained here concerning correctness and functionality must always be taken into account. Despite this fact, we compare the performance of these approaches as all of them implement a priority queue, albeit with different characteristics.

3 Performance Tests

Priority queues are often used at the core of schedulers, such as in the Tempo project. In such an environment, the performance of the queue is a critical factor in the overall system. Scalability in the number of threads is another property that is crucial for concurrent algorithms. Both these parameters are measured in the performance tests.

All of the algorithms have been implemented in Java, and the performance tests are executed on a Java 6.0 runtime environment on a Linux SMP system.

3.1 Metrics

A key attribute of every concurrent algorithm is its scalability in the number of threads. Two cases are considered. In a fully concurrent environment, each thread is executed on a CPU core. In time-sharing mode, more threads than CPU cores are available, and the operating system’s scheduler governs access to the CPU cores in a time-sharing mode. For the performance tests, an 8-way multi-core machine was used. This allows scalability to be measured in a fully concurrent environment from 1 to 8 threads and in a time-sharing environment for more than 8 threads.

Another parameter is the access pattern of the priority queue. Each thread puts an item with random priority into the queue or gets an item from the queue, and after that does some “local” work. The amount of local work determines the contention level and is used to simulate realistic

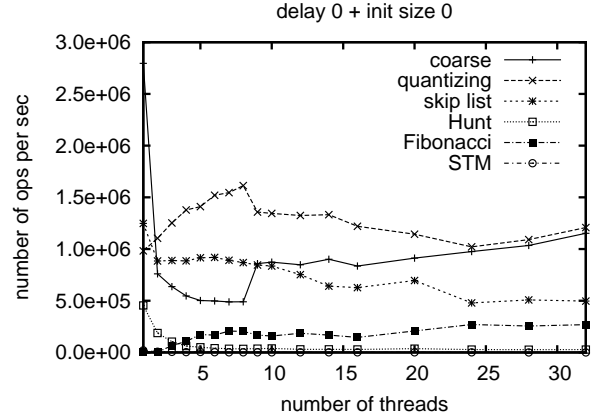


Figure 1. Throughput for the very high contention test case without local work and an initially empty queue

application behavior. At maximum contention level no local work (“working delay”) is done. For a more realistic behavior the time spent in local work varies from 1 μ s to 1 ms with exponential stepping. We simulate this delay by executing integer computations in between queue operations. The computations are based on random input and an iterative computation that cannot be optimized by a compiler.

As a last parameter, we vary the initial queue size, starting with an empty queue or a queue pre-loaded with 100 or 10000 elements. Queues with initially 10000 elements we call “big queues”.

The results presented in the figures below show the accumulated number of operations of all threads including the local work done after each operation. So if there is a “working delay”, we measure the performance of the entire application and not only of the operations on the queue.

3.2 Results of Selected Scenarios

In order to compute confidence intervals, we assume that the cumulative throughput for a given scenario follows a normal distribution for repeated test runs. The bounds of the confidence intervals for a confidence coefficient of 95% differ by less than 3% from the computed average for scenarios where the queue was initially empty or pre-loaded with 100 elements only. For bigger queues, especially for the quantizing queue and the STM-heap, they differ by about 6%. An exception is the STM-heap, the bounds differ by as much as 10% from the average.

First, we test the very high contention case without any simulated local work outside operations on the queue and an initially empty queue. The results are shown in figure 1. Recall that we have 8 physical CPU-cores.

The quantizing queue and the parallel Fibonacci heap are the only queues scaling in this case for up to 8 threads.

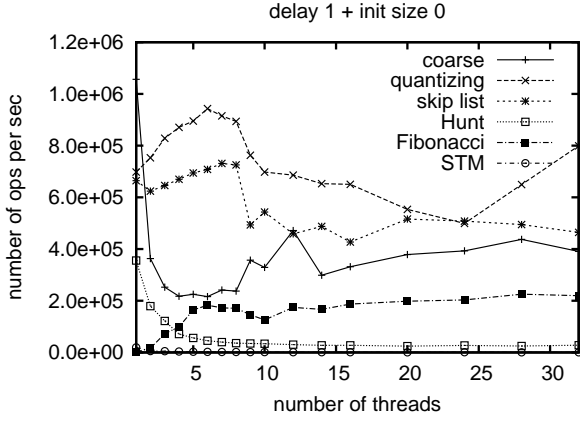


Figure 2. Throughput for the high contention test case with short local work (roughly 1 μ s) and an initially empty queue

Regarding performance, Sundell’s and Tsigas’ lock-free skip-list exhibits a competitive performance, but the quantizing queue outperforms all other implementations. The Hunt heap shows low scalability at this level of contention, whereas the STM-based heap obviously implies very much overhead due to the DSTM2 factory. The highest performance is achieved by the coarse-locked heap running single-threaded. The drastic break-down of the performance from 1 to 2 threads is due to the implementation of locking in Java, which distinguishes thin and thick locks. The principle of these Java locks is explained in detail in [2].

Another evident pattern is the jump of the number of operations in the coarse-lock implementation from 8 to 9 threads. This can be explained by the CPU-hopping effect, which is caused by the process scheduler in the kernel, because it attempts to do load-balancing between CPU cores. As all other CPU cores are idle because the threads running on them are waiting for the global lock, the scheduler reschedules the only working thread to another CPU core, which causes additional cash-trashing. This effect is reduced by pinning threads to CPU cores; something that we didn’t do for our tests.

When running in timesharing mode, with more than 8 threads, the behavior of the algorithms no longer follows an easily explainable pattern. However, the relative performance between the algorithms remains stable (c.f. Figures 1, 2). The exception is the coarse-lock implementation in the high contention scenario (Figure 1) that outperforms the lock-free skip-list when using more than 10 threads.

Figure 2 shows the results for the test scenario with short local work (1 μ s) and an initially empty queue. Here, the lock-free skip-list shows even better performance in relation to the other implementations, but the quantizing queue still outperforms all of them. The coarse-locked queue shows a worse performance than in the scenario described above, es-

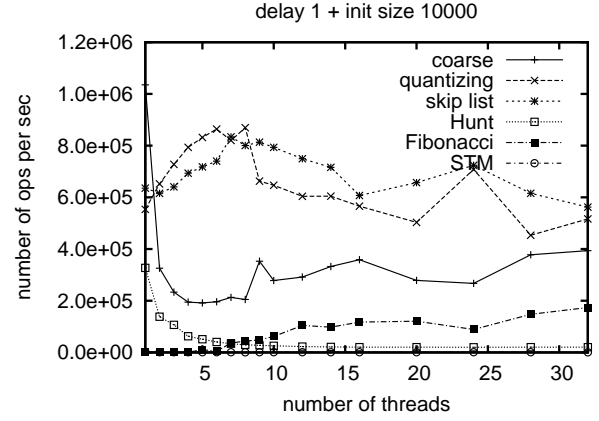


Figure 3. Throughput for the test case with short local work and a queue initialized with 10000 elements

pecially for a lower CPU core/thread ratio. However, again the best performing test case with a lower dominance than before is the single-threaded coarse-lock case.

When pre-loaded with 10,000 random elements, the lock-free skip-list, the coarse-locked heap and the Hunt heap do not show significant deficits, whereas the quantizing queue and the Fibonacci heap suffer a noticeable loss of performance (Figure 3).

3.3 Results Summary

There are also scenarios with significant local work tested that, as expected, show advantages for the coarse-locked heap for up to 8 threads. The Hunt heap and the Fibonacci heap catch up the other implementations because all performances have been relativized. But we can still see the same picture with the quantizing queue compared with the lock-free skip-list: for big queues (10,000 elements), the skip-list has a higher throughput, whereas the quantizing queue performs better than the skip-list on small queues.

Tables 1 and 2 provide a rough summary of our survey with the coarse-locked queue (CLQ), Hunt heap (HH), parallel Fibonacci heap (PFH), lock-free skip-list (LFSL), quantizing queue (QQ), and the STM-based queue (STMQ). The terms *very low* (v. low), *below/above average* (b./a. avg), *high*, etc. for characterizing the scalability and performance are used relatively among the implementations studied here. The term *high contention* (HC) denotes short local work (1 μ s and less) and small queue scenarios while the term *low contention* (LC) refers to significant local work (more than 10 μ s) and big queue scenarios. Table 1 summarizes the results concerning performance and scalability for fully concurrent tests. Table 2 gives an overview of the performance results in timesharing mode, the variance of operations per thread per second and the correctness condi-

Table 1. Summary of the results for a CPU-core/thread-ratio ≥ 1 (performance and scalability)

Method	Perf. with HC	Perf. with LC	Scale. with HC	Scale with LC
CLQ	avg	v. high	v. low	high
HH	low	avg	v. low	low
PFH	b. avg	b. avg	v. high	avg
LFSL	high	high	a. avg	high
QQ	v. high	avg	high	low
STMQ	v. low	low	avg	low

tions each queue implementation fulfills (linearizable (lin), sequentially consistent (s.c.), random (rand)).

A surprising result is that the coarse-grained locking approach with one thread achieves the highest absolute performance in the high-contention scenarios, where no or very little additional work is done besides the queue operation. In this scenario, the costs of concurrency control outweighs the benefits. As expected, however, the single-lock approach does not scale at all in a high-contention scenarios. For low-contention, fully concurrent (up to 8 threads) scenarios, the coarse-locked case performs better than other solutions. It is particularly noticeable that for higher numbers of threads, the lock-free approaches perform well.

Another interesting value is the variance of the number of operations per thread and per second in relation to the average of each implementation. The Hunt heap has the lowest variance with only 10-20% of the average value. This means that the number of operations of each thread does not vary much in a designated interval. The Fibonacci heap also has a reasonable variance, with only 25-30%. The lock-free skip-list and coarse-locked heap vary on average with 40%. The quantizing queue and the STM-based heap show a relatively high variance, with about 100%. The higher the variance is, the less reliable is an a priori estimation of how many operations will be done in a time interval. An a priori estimation is of special interest for real-time applications, which need to know the behavior of its components.

3.4 Use Case Results

We also study the behavior of the three most competitive priority queues in the context of our publish/subscribe system *Tempo* (see Figure 4). We run the tests on the same machine as the previous tests, with the publishers and the subscribers running on the same system. In the figure we observe that for up to 8 threads we clearly have a better performance for the lock-free queues than for the coarse-locked queue. As soon as timesharing mode is entered, the

Table 2. Summary of the general results and characteristics (correctness condition, variance of operations per thread per second, performance)

Method	Corr. cond.	Variance	Perf. LC c/thr < 1	Perf. HC c/thr < 1
CLQ	lin	avg	avg	avg
HH	lin	v. low	low	low
PFH	rand	low	b. avg	b. avg
LFSL	lin	avg	v. high	high
QQ	s.c.	high	high	v. high
STMQ	lin	high	v. low	v. low

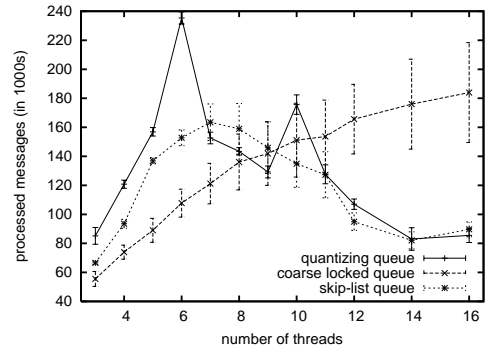


Figure 4. Message throughput within the *Tempo* messaging system with confidence interval bars

performance of the publish/subscribe engine decreases significantly for the lock-free approaches, whereas the scenario using the coarse-locked queue shows better performance. There are many factors in the pub/sub engine that influence the results of this test beside the priority queue. A discussion of these factors would, however, exceed the scope for this paper.

4 Related Work

Shavit and Zemach [16] report performance tests using a simulated distributed-shared-memory machine for the coarse-lock queue, the Hunt heap and the quantizing queue, among others. The tests with up to 16 threads are comparable to the tests done in this paper. It also used an initially empty queue, and each thread conducted an unspecified “small” amount of local work. While the results for the quantizing queue agree, the coarse-grained lock approach is the worst performer in Shavit and Zemach’s experiment. However, we find it to be in the middle field, and in some scenarios even being one of the best.

Hunt et al. [11] use a fixed set of operations for

their experiment, either insertion-only, deletion-only, or a insertion/deletion-pair test. They compare their fine-grained locking with the traditional coarse-locking approach, where each process conducts “significant real work” between operations on the queues. Their experiments with empty or small initial queue size are comparable to ours and also match the result in that the coarse grained locking approach outperforms the fine-grained locking approach. Hunt et al. show that fine-grained locking outperforms coarse grained locking for queue sizes that are larger than 100,000 elements. Note that we did not conduct tests for queues with more than 10,000 elements.

Sundell and Tsigas [17] compare their skip-list implementation with Hunt’s algorithm on machines ranging from 2 to 64 processors. In line with our results, they find that Hunt’s algorithm exhibits worse performance than the skip-list implementation, both with respect to absolute number as well as scalability in the number of threads. However, we cannot confirm the level of scalability reported for the skip-list. We observe a scalability that is noteworthy but not outstanding.

5 Summary and Conclusion

Using the example of a concurrent priority queue, we have investigated different approaches to concurrency control, i.e. traditional lock-based approaches and lock-free algorithms as well as an algorithm based on the software transactional memory model. We have characterized the properties and discussed the correctness condition of each algorithm. In addition, we have evaluated the performance of the algorithms using several benchmarks that were carried out on a eight-core machine. To the best of our knowledge, this performance discussion is wider and compares more implementation techniques in a homogeneous environment than what has been done before.

We find that the highest performance is achieved by the coarse-locked binary heap when executed by a single thread, i.e. when the costs for concurrency control are negligible due to the thin-locks used by Java. When accessed by multiple threads, the quantizing queue followed by the lock-free skip-list provide the best performance. While the quantizing queue shows good scalability, it also has a higher variance and is, in contrary to the skip-list, not linearizable.

Finally, we also observe that the Linux process scheduler has a significant effect on the performance. In most cases, the CPU-hopping effect has a higher impact on the performance than the actual implementation itself i.e. pinning threads to CPU cores yields significant improvements in performance.

References

- [1] R. Ayani. LR-algorithm: Concurrent Operations on Priority Queues. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 22–25, Dec. 1990.
- [2] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, June 1998.
- [3] D. Bauer, L. Garcés-Erice, and S. Rooney. Tempo: A Simple Time-Sensitive Messaging System. In *1st IEEE Workshop on Autonomic Communications and Network Management (ACNM’07)*, pages 1–8. IEEE Press, May 2007.
- [4] K. Dragičević and D. Bauer. Survey of Concurrent Priority Queue Algorithms. Technical Report RZ 3700, IBM Research, Nov. 2007.
- [5] R. Ennals. Software Transactional Memory Should Not Be Obstruction-Free. Technical Report IRC-TR-06-052, Intel Research Cambridge, Jan. 2006.
- [6] K. Fraser. Practical Lock-Freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Feb. 2004.
- [7] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *ICDCS’03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522. IEEE Computer Society, May 2003.
- [8] M. Herlihy, V. Luchangco, and M. Moir. A Flexible Framework for Implementing Software Transactional Memory. In *OOPSLA’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 253–262. ACM Press, Oct. 2006.
- [9] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [10] Q. Huang. A Evaluation of Concurrent Priority Queue Algorithms. Technical Report MIT/LCS/TR-497, Massachusetts Institute of Technology, Feb. 1991.
- [11] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Information Processing Letters*, 60(3):151–157, Sept. 1996.
- [12] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computer*, 28(9):690–691, Sept. 1979.
- [13] M. M. Michael and M. L. Scott. Simple, Fast and Practical Non-Blocking Concurrent Queue Algorithms. In *PODC’96: Proceedings of the 15th Annual ACM Symposium on Principles of distributed computing*, pages 267–275. ACM Press, May 1996.
- [14] W. N. Scherer III. and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, pages 70–79, Apr. 2004.
- [15] N. Shavit and D. Touitou. Software Transactional Memory. In *PODC’95: Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM Press, Aug. 1995.

- [16] N. Shavit and A. Zemach. Scalable Concurrent Priority Queue Algorithms. In *PODC'99: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 113–122. ACM Press, May 1999.
- [17] H. Sundell and P. Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In *IPDPS '03: Proceedings of the 17th International Parallel and Distributed Processing Symposium*, pages 609–627. IEEE Press, May 2003.