Per-Thread Batch Queues For Multithreaded Programs

Tri Nguyen, M.S.
Robert Chun, Ph.D.

Computer Science Department
San Jose State University
San Jose, California 95192

Abstract

Sharing resources leads to contention in multithreaded programs. In order to safely access shared resources, such as queues, threads must serialize access to them. One common way to serialize resource access is to use programming language-supported locks. However, serializing threads on highly contended resources commonly leads to poor performance due to excessive context switching or busy-waiting. Specifically in multithreaded servers, where a dispatcher thread dispatches jobs to a worker thread, serializing access to a shared work queue leads to performance bottlenecks. This paper discusses the problem caused by thread synchronization on shared queues, related work addressing this problem, a new per-thread batch queue algorithm, experimental results comparing batch and non-batch queues, and an analysis of the results.

Keywords: Software Performance, Parallel Algorithms, Synchronization, Data Structures, Multithreaded.

1. Introduction

A thread is a section of code executing independently of other threads within the same process [7]. There are many advantages in developing multithreaded software. One reason is to better utilize system resources by assigning independent tasks to separate threads. While one thread is context-switched out waiting on I/O, another thread can context-switch in making use of the processor. The entire program does not come to a halt because one thread is waiting on I/O or another condition.

Since several threads can exist inside of a process and share that process's context, a thread can be thought of as being a "lightweight process". Executing in the same context, race conditions occur when two or more threads access shared data concurrently. Without a synchronization protocol on shared data, data inconsistencies can occur due to race conditions. Traditional synchronization protocols involve serializing access to critical sections by grouping all steps in the critical section into one atomic transaction through the use of software locks.

1.1 Problem Addressed

Multithreaded servers take advantage of multiprocessor machines by running a thread on each processor. One idea is to design the server with a dispatcher thread dispatching work to multiple worker threads via a work queue. A dispatcher thread can dispatch work by enqueueing work on a shared queue and a worker thread can get work by dequeueing from the shared queue.

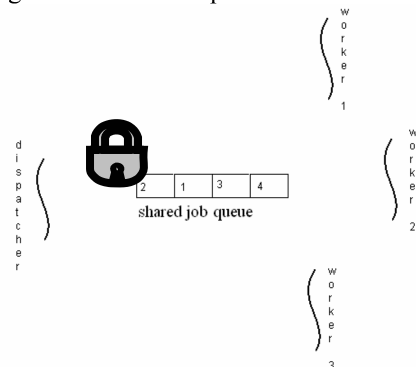

Figure 1. Dispatcher & Worker Threads Sharing Job Queue

To keep the shared queue in a consistent state, threads serialize access to the shared queue. If the queue is highly contended, serialization using condition variables and busy-wait locks can lead to performance bottlenecks. Using condition variables, when a thread, say ThreadA, must wait for a condition (queue not empty) on the shared queue, it will context-switch giving up the processor to another thread. Another thread, say ThreadB, may also need to access the queue only to find it in a state not satisfying ThreadB's condition. ThreadB must then context-switch. Using condition variables on highly contended queues leads to context-switch thrashing because a condition a thread is waiting on is not met. Not only does a thread context-switch, but it must also acquire and release a lock.

If busy-waiting is used, a thread keeps spinning on a processor, not doing any useful work constantly acquiring and reacquiring locks while waiting for a condition. Serialization leads to excessive context-switching or threads spinning on processors.

This paper will discuss various algorithms to improve system performance when threads access highly contended shared queues. The focus of the experiments will compare a new per-thread batch algorithm versus its non-batch counterpart. This paper will also compare the performance of per-thread queues versus single queue algorithms.

2. Related Work

2.1 Busy-waiting

Known as busy-waiting [6, 7, 11], one solution is to have a thread spin on a processor, waiting for a condition. While spinning, the thread acquires and releases a lock during each iteration. Because a thread spins while waiting for a condition (queue has work, queue is not full), a processor is under-utilized and not really doing any useful work.

2.2 Condition Variables

Although busy-waiting serializes access to shared data, the busy-waiting algorithm spins taking up processor time until a condition is met. The main disadvantage of busy-waiting is that it wastes CPU resources. Condition variables [6, 11] solve this problem by putting the thread to sleep (and atomically releasing the lock) when a condition prevents the thread from making the progress it wants. The operating system will signal the blocked thread when the condition is met, waking up the thread. Condition variables allow another thread to execute on a processor that would otherwise be spinning.

2.3 Non-blocking Lock-free Queues

Michael and Scott [5] present a non-blocking lock-free queue algorithm based on an atomic hardware supported compare_and_swap (CAS) instruction instead of any language-supported locks. CAS takes three arguments, "the address of a shared memory location, an expected value, and a new value" [5]. If the memory location has the expected value, then the new value is written to the memory location and the function returns true. Otherwise, the function returns false. The Michael and Scott algorithm outperforms other queue algorithms [5]. Michael and Scott's non-blocking lock-free queue has been incorporated into Sun Microsystems® Java™ Development Kit 1.5.

For each enqueue or dequeue operation, the algorithm does a series of reads, obtaining head or tail values from the queue, and then uses the CAS instruction to perform the required queue manipulation to either insert or remove an element. If the CAS is successful (returns true), then the queue operation is successful. Otherwise, the algorithm loops trying again. Each time the algorithm has to loop again, it only means another thread completed an enqueue or dequeue operation invalidating the earlier reads of the current thread. At least one thread is guaranteed progress.

This algorithm is lock-free since software locks are not used. With the critical section "unlocked", multiple threads concurrently access the "critical section". Updates to the queue are safely checked during the CAS operation.

2.4 Two-lock Queues

Two-lock queues [5, 9] have two locks, the tail lock and the head lock. The tail lock is acquired when manipulating the tail pointer while the head lock is acquired for manipulating the head pointer. Initially, both head and tail pointers point to a dummy node. The dummy node forces enqueuers to "never have access to [the] head and dequeuers to "never have to access [the] tail" [5]. If the contents of the queue only contains the dummy node, the queue is considered empty. When enqueueing, the tail lock is acquired and then the new node is inserted to the tail of the queue. When dequeueing, the head lock is acquired and the head node is removed.

2.5 Thread-Specific Storage Pattern

Schmidt and Pryce [8] describe a mechanism to access thread-specific data without using locks. When a thread needs to access its private data, it passes a thread-specific key to the TS Object Proxy which retrieves the thread specific data from a TS Object Collection. The drawback of this scheme is that multiple threads cannot share data since each thread has a unique key.

3. Per-Thread Batch Queues

Per-thread batch queues are designed to (1) reduce contention and (2) reduce the number of accesses to the shared queue. First, we present batch queues in which jobs are enqueued and dequeued in batches (N jobs per operation). Then, we present per-thread queues in which each worker thread has its own queue. Finally, we present a combined approach, namely per-thread batch queues, in which each worker thread has its own batch queue.

3.1 Batch Queues

Highly contended shared queues lead to higher levels of thread context-switching or busy-waiting depending on the synchronization scheme. Serialization restricts the number of accesses to the shared queue per unit of time. If each access does more work, then context switching or processor spinning is lowered. A batch queue algorithm can lower contention by lowering the rate at which threads have to access the queue while enqueueing and dequeueing more queue elements. While traditional algorithms enqueue and dequeue single elements, batch queues enqueue and dequeue a batch each time. The batch queue algorithm is ideal for high-load and highly contended queues. Figure 2 illustrates the structural difference between traditional single-element queues and batch queues.

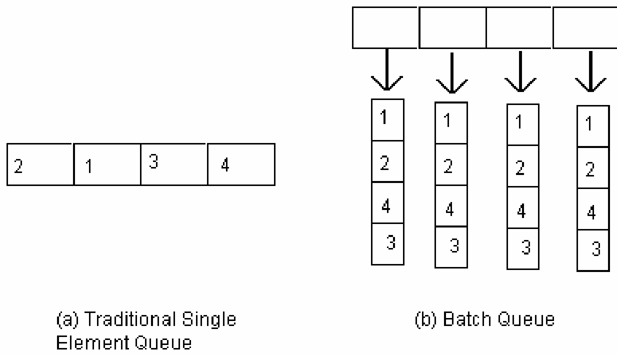

Figure 2. (a) Single Element Queue and (b) Batch Queue

Batch queues require fewer operations to process the same number of elements compared to single element queues. The number of operations is defined as the number of successful attempts to enqueue or dequeue items and not the number of total attempts where some attempts can be unsuccessful. The formula below defines the tight bound for number of operations required between batch and non-batch algorithms.

Let b $\leftarrow$ batch size

Let N $\leftarrow$ number elements to operate on

Let $\theta(N)$ $\leftarrow$ number of operations required to operate on N elements

For non-batch queues, $\theta(N) = 2N$.

For batch queues, $\theta(N) = (2N)/b$.

Batch queues require fewer operations than non-batch queues by a factor of b.

3.2 Per-thread Queues

To further minimize contention on the shared queue, each worker thread can have a private queue shared with only the dispatcher thread. In the traditional dispatcher-worker thread model, all threads contend for one queue. By allocating one queue per worker thread, contention is only between one worker thread and one dispatcher thread per queue. Worker threads no longer contend with each other.

3.3 Per-thread Batch Queues

To take advantage of both the batch queue and per-thread queue algorithm, a combined algorithm can be used to lower contention further. In the combined approach, each worker thread will have a private batch queue shared between the worker thread and the dispatcher thread.

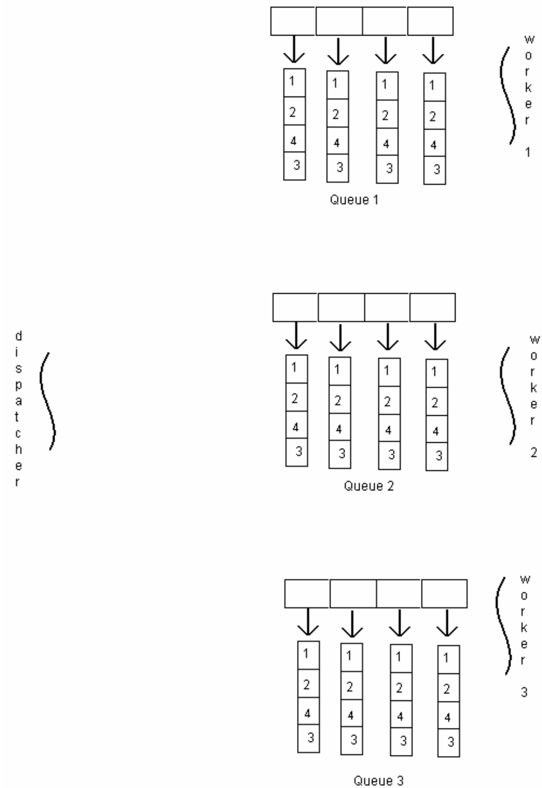

Figure 3. Per-Thread Batch Queue

4. Experimental Results

4.1 Related Experiments

Other researchers [5, 12] measure the throughput of their respective synchronization algorithms on experiments inserting and removing elements from a shared data structure on multiprocessor machines. Michael and Scott [5] process one million elements using a shared queue on a 12 processor machine. Sundell and Tsigas [12] process 10,000 * N elements, where N is the number of threads using a shared priority queue on single, 4, and 64 processor machines. Common to both experiments is that the experiments run on a single machine, looping, processing elements on and off a shared data structure. Other researchers' experiments do not manage client connections.

4.2 Clientless Experiments

The design of the clientless experiments is based on related experiments of other researchers [5, 12] where the experiments run on a single multiprocessor machine without any clients connecting to a server process. The server process consists of a dispatcher thread looping, generating, and dispatching jobs onto each worker thread's queue. Each worker thread dequeues jobs from its queue and processes the job. The processing time for each job will vary depending on the job size set by each experiment.

With 4 processors, a single dispatcher and 3 worker threads were instantiated. Four synchronized queueing algorithms (non-blocking lock-free queue, two-lock queue, condition variables, and busy-waiting) and its batch counterpart were implemented and tested using JDK™ 1.5. Varying the job size, running times for each thread, CPU usage, and context switching activity were measured. To gain a better understanding of the behavior of each algorithm, Hewlett Packard's HPJMeter™ was used to profile the execution time of each method call per algorithm. Microsoft Performance Monitor™ measured context switching activity. For notation purposes, each algorithm is abbreviated as follows:

CLQ: ConcurrentLinkedQueue
TLQ: Two-lock Queue
CVQ: Condition Variable Queue
BWQ: Busy-wait Queue

An abbreviation with a "B" prefix before any of the above abbreviations is associated with that algorithm's batch version such as BTLQ means Batch Two-lock Queue.

4.2.1 Single Element Queue with Empty Jobs

The parameters for the base case clientless test are:

- Job size ← 0
- Batch size ← 50
- Number of batches processed ← 100,000

The base case test has all threads sharing a single queue. Each job will have a size, N, where it will loop N times when it is called to do work. The dispatcher generates a batch of jobs, but enqueues one job at a time in the single element queue. Table 1 shows the resulting average running times for all threads (1 dispatcher and 3 workers) and average context switches (CS) per second for each algorithm.

Algorithm	Average running time (ms)	CS per sec
CLQ	26969	838
TLQ	68226	11065
CVQ	343985	116434
BWQ	104078	324

Table 1. Results for Base Case Clientless Test

4.2.2 Per-thread and Per-thread Batch Queue with Empty Jobs

The second clientless experiment sets the job size to zero. Table 2 shows the results for the first clientless experiments where:

- Job size ← 0
- Batch size ← 50
- Number of batches processed ← 100,000

Algorithm	Average running time (ms)	CS per sec
CLQ	26969	865
BCLQ	13344	1860
TLQ	34843	962
BTLQ	13406	1945
CVQ	175094	107403
BCVQ	7047	32336
BWQ	29265	957
BBWQ	13500	1944

Table 2. Clientless Experiment with job size set to 0

Per-thread queue algorithms show significant performance gains when compared with single-queue algorithms. Comparing running times in Table 1 (single-queue) with that in Table 2 (per-thread queue), the running time ratios are as follows:

- Per-thread CLQ ties single-queue CLQ 1:1
- Per-thread TLQ beats single-queue TLQ by 1.9:1
- Per-thread CVQ beats single-queue CVQ by 1.9:1
- Per-thread BWQ beats single-queue BWQ by 3.5:1

Comparing the running times for the batch and non-batch algorithms, the batch version processes jobs at a faster rate. In particular, the running time ratios are as follows:

- BCLQ beats CLQ by factor of 2:1
- BTLQ beats TLQ by factor of 2.5:1
- BCVQ beats CVQ by factor of 24.8:1
- BBWQ beats BWQ by factor of 2.1:1

There is a glaring discrepancy between the average thread running time of the CVQ algorithm relative to other algorithms. There is, for the most part, a pattern associating running times with context switching activity: the higher the context switching activity, the longer the running time. CVQ algorithm's slower running time can be attributed to the much higher level of context switching activity at 107403 context switches per second versus 32336 (BCVQ) to the next runner up, and 1945 (BTLQ) to the third runner up. Perhaps the reason CVQ takes the longest is because a context switch is more expensive than a function call into the JVM (CAS-based synchronization). A context switch involves kernel overhead on top of a function call such as saving and restoring thread states.

4.2.3 Profiled Per-thread Batch Queue

While condition variable algorithms produce high levels of context switching, busy waiting algorithms produce high levels of enqueue and dequeue attempts. Each attempt can fail resulting in a retry. High levels of enqueue and dequeue attempts can result in a slower running time than high levels of context switching. The next experiment turns on Java's profiling option (-prof) enabling recording of running times and method call counts into a profiling file. HPJmeter™ can then read the profiling file and produce a graph, ranking method call counts and running times (virtual cpu cycles). Parameters for profiled experiment:

- Job size ← 10
- Batch size ← 50
- Number of batches processed ← 1,000

Context switching is not necessarily the worse side effect. Although BCVQ context switches more than other batching algorithms, it outperforms them (Table 2). A closer look at the profiled data shows that the other busy waiting algorithms attempt more enqueues and dequeues that fail more often, resulting in higher running times.

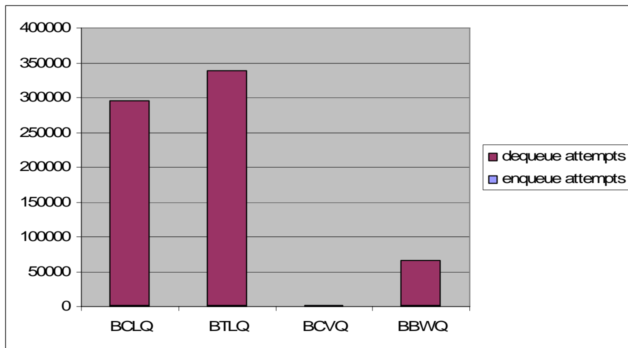

Figure 4. Enqueue and Dequeue Attempts

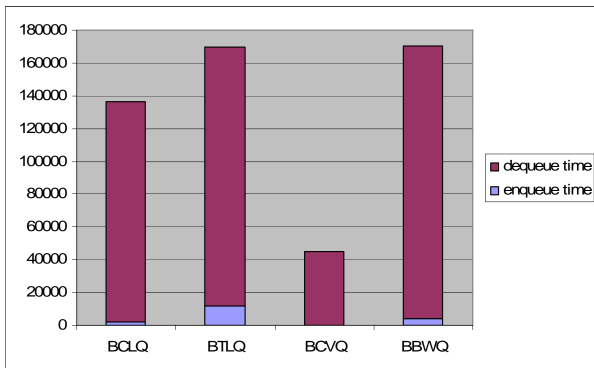

Figure 5. Virtual CPU Cycles

Figures 4 and 5 show the correlation between number of attempts to virtual CPU cycles per algorithm**.** In general, for batched algorithms, a higher number of enqueue and dequeue attempts results in a longer running time.

4.3 Client Server Experiments

The design of the client server experiments consist of client machines creating client threads sending batches of job requests to the server. The server process consists of a dispatcher thread dispatching jobs onto each worker thread's queue. Each worker thread dequeues jobs from its queue and processes the jobs. After processing the batch of job requests, the worker thread sends an acknowledgement to the client that the jobs have been processed.

With 1 dispatcher thread running on a quad processor, 3 worker threads are created to run on the other 3 processors. Using laptops with AMD® Athlon™ XP 2000+ processors and 256MB RAM as client machines, client threads looped sending requests to the server. Each request serviced a batch of jobs. Average running times for client threads, server CPU usage, and context switching activity were measured.

4.3.1 Client Server Experiment 1: Short Jobs

For Client Server Experiment 1, the job size is set to 10 and the batch size is set to 150. On each of the 3 laptops, 300 client threads were created with each thread looping 80 times sending a request to process a batch of jobs each time. With a smaller batch size than 150, batch queues performed relatively the same as non-batch queues but BCVQ did beat CVQ by a factor of 5.6. Increasing the batch size to 150, lock-free algorithms experienced a better performance gain.

Algorithm	Average client thread running time (ms)	Context Switches per sec (server)
CLQ	95541	570
BCLQ	48654	985
TLQ	62174	998
BTLQ	48919	1026
CVQ	387753	101720
BCVQ	41922	2835
BWQ	65764	850
BBWQ	48335	939

Table 3: Results for Client Server Experiment 1

Comparing the running times for the batch and non-batch algorithms, the batch version processes jobs at a faster rate, resulting in faster average finish times of client threads. In particular, the average running time ratios are as follows:

- BLCQ beats CLQ by factor of 1.9:1
- BTLQ beats TLQ by factor of 1.2:1
- BCVQ beats CVQ by factor of 9.2:1
- BBWQ beats BWQ by factor of 1.3:1

Like the clientless experiments, there is a glaring difference in performance between the BCVQ:CVQ ratio versus the other batch:non-batch ratios. System performance is correlated with, for the most part, context switching.

Although BCVQ has a higher server context switching activity level than other batch algorithms, it has the faster client running time. The CAS-based busy-waiting algorithms may have much higher levels of unsuccessful enqueue and dequeue attempts.

4.3.2 Client Server Experiment 2-Profiled Test

Client Server Experiment 2 profiles the server using HPJmeter™ examining enqueue and dequeue attempts as it correlates to virtual CPU cycles. From Table 4, for the most part, the batch algorithms have fewer enqueue and dequeue operations and as a result, take less virtual CPU cycles when trying to enqueue and dequeue jobs. The only exception is the correlation between attempts and virtual CPU cycles when comparing CVQ and BCVQ. Although BCVQ has less attempts, it has a higher number of virtual CPU cycles. This could be the result of the profiling tool slowing time down so that the worker thread performs work on longer jobs, accessing the job queue at a slower rate (compare running times of Table 2, 100,000 batches processed, and Figure 5, 1,000 batches processed). Accessing the job queue at a slower rate leads to less contention which does not favor the batch queue algorithm. The parameters for this experiment are as follows:

- Job size ← 10
- Batch size ← 150
- Number of client threads ← 50
- Each client sends 10 batch requests

Algorithm	Enqueue	Dequeue
CQ	75450	719663
BCLQ	503	463267
TLQ	75450	3783405
BTLQ	503	483931
CVQ	75450	75003
BCVQ	503	503
BWQ	220644	477456
BBWQ	990	94426

Table 4. Enqeue and Dequeue Attempts

The batch:non-batch combined enqueue and dequeue ratio show batch queues having fewer attempts.

- BCLQ:CQ is 1:1.7
- BTLQ:TLQ is 1:7.9
- BCVQ:CVQ is 1:149
- BBWQ:BWQ is 1:7.3

The result is fewer enqueue and dequeue attempts (especially for the context switching activity of CVQ and BCVQ), and faster virtual CPU running times. Table 5 shows the virtual CPU cycles per algorithm.

Algorithm	enqueue	dequeue
CQ	152270	446387
BCLQ	868	210464
TLQ	881564	2047711
BTLQ	5986	223833
CVQ	9647	65908
BCVQ	79	80353
BWQ	367961	1091898
BBWQ	2048	237861

Table 5. Virtual CPU Cycles

5.0 CONCLUSION

Without batching, per-thread queues show significant performance gains compared to single-queue algorithms. In the case of per-thread CLQ versus single-queue CLQ, there is no measurable difference in performance. Per-thread TLQ, CVQ, and BWQ beats single-queue TLQ, CVQ, and BWQ by a factor of 1.9, 1.9, and 3.5, respectively. Per-thread batch queues can significantly improve job throughput by minimizing synchronization activity whether it be (1) context switching through the use of condition variables or (2) CAS-based synchronization. Condition variable algorithms get the most performance gain when per-thread batch queues are applied. As shown in Experiment 2 (clientless and job size is set to 0), a 24.8:1 performance gain is achieved when per-thread batching is applied to condition variables. In the same experiment, CAS-based synchronization algorithms experience about a 2:1 performance gain. Other clientless and client-server results show performance gains when per-thread batch queues are applied. In the case when jobs are long, batch queues can perform worse than non-batch queues. In the client-server experiments, per-thread batch queues lead to less of a performance gain. Managing clients leads to less contention for the job queue so the synchronization activity involved in non-batch queues may not be that much higher than batch queues.

CVQ is the worst performer because excessive context switching is more expensive than excessive CAS-based calls into the JVM. However, some level of context switching is healthy as demonstrated by BCVQ's lower running time than other batch CAS-based algorithms. Although CAS-based algorithms context switch much less, they spin on a processor when their attempts to update the queue are unsuccessful. CAS-based algorithms spin until the queue operation is successful. Spin-waiting leads to processor under-utilization because another thread could be doing useful work on that processor. CAS-based spin-waiting algorithms lead to more failed attempts and processor under-utilization, while using condition variables lead to expensive context-switching but less failed attempts.

6.0 FUTURE WORK

The work presented in this report shows per-thread batch queues outperforming single-queue and non-batch queues. However, there are weaknesses and possible improvements to be considered beyond the work presented in this report. Batch queues perform slower than non-batch queues for longer jobs except when using condition variables. Although technology will build faster processors making today's longer jobs shorter tomorrow, work beyond processor-intensive tasks will slow down even the shortest jobs. These tasks include client-server communication, file input/output, and database queries.

Although context switching is very costly resulting in the slowest algorithm, CVQ, some level of context switching is healthy (BCVQ) when compared to spin-waiting algorithms. Further investigation can examine and suggest healthy levels of context switching and spin-waiting activity. Perhaps a hybrid solution coupling lock-free spin-waiting and context-switching is the solution. Furthermore, it would be useful to know how long (time and attempts) a thread should spin before context-switching.

7.0 REFERENCES

[1] Anderson, Tom. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems. Volume 1, Number 1, January 1990.

[2] Herlihy, M., Shavit, N. Introduction to Multiprocessors: Algorithms, Data Structures, and Programming: Room Synchronization. Retrieved from http://www.cs.brown.edu/courses/cs176/room.pdf on December 2, 2004.

[3] Lindholm, T., Yellin, F. The Java™ Virtual Machine Specification, Second Edition. Addison-Wesley Professional. April 14, 1999.

[4] Magnusson, P., Anders, L., Hagersten, E. Efficient Software Synchronization Large Cache Coherent Multiprocessors. SICS Research Report T94:07. February 1994.

[5] Michael, M., Scott, M. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. 15th Symposium on Principals of Distributed Computing, May 1996.

[6] Oaks, S., Wong, H. Java Threads. O'Reilly , 1999.

[7] Silberschatz, A., Galvin, P. Operating System Concepts. Addison-Wesley Publishing Company, 1994.

[8] Schmidt, D., Pryce, N. Thread-Specific Storage for C/C++: An Object Behavioral Pattern for Accessing per-Thread State Efficiently. 4th Annual Pattern Languages of Programming Conference, September 1997.

[9] Sun Microsystems' ® JDK 1.5 java.util.concurrent Package. Retrieved from http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html on November 15, 2004.

[10] Sundell, H., Tsigas, P., Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. Proceedings of the International Parallel and Distributed Processing Symposium. April 2003.

[11] Tanenbaum, Andrew. Modern Operating Systems. Prentice Hall, 1992.

[12] Valois, John. Implementing Lock-Free Queues. Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems. October 1994.