Liberty Queues for EPIC Architectures

Thomas B. Jablin[1] Yun Zhang[1] James A. Jablin[2] Jialu Huang[1] Hanjun Kim[1] David I. August[1]

[1]Department of Computer Science, Princeton University

[2]Department of Computer Science, Brown University

{tjablin,yunzhang,jialuh,hanjunk,august}@princeton.edu jjablin@cs.brown.edu

Corresponding Author: Thomas B. Jablin

Liberty Queues for EPIC Architectures

Thomas B. Jablin Yun Zhang James A. Jablin[2] Jialu Huang Hanjun Kim David I. August

Department of Computer Science, Princeton University [2]Department of Computer Science, Brown University

{tjablin,yunzhang,jialuh,hanjunk,august}@princeton.edu jjablin@cs.brown.edu

Abstract

Core-to-core communication bandwidth is critical for high-performance pipeline-parallel programs. Hardware communication queues are unlikely to be implemented and are perhaps unnecessary. This paper presents Liberty Queues, a high-performance lock-free software-only ring buffer, and describes the porting effort from the original x86-64 implementation to IA-64. Liberty Queues achieve a bandwidth of 500 MB/s between unrelated processors on a first generation Itanium 2, compared with 281 MB/s on modern Opterons and 430 MB/s on modern Xeons claimed by related works. We present bandwidth results for seven different multicore and multiprocessor systems, as well as a sensitivity analysis.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms algorithms, performance, experimentation

Keywords smtx, nonblocking synchronization, lock-free, queue, streaming instructions, pipeline parallel, multicore, multiprocessors

1. Introduction

Pipeline parallelism has achieved scalable performance on general purpose applications previously thought difficult or impossible to parallelize [1]. However, these parallelizations run on simulated Itanium processor modified to support synchronization arrays [14]. In the simulation, core-to-core communication via synchronization arrays was as cheap as a load or store to L2 cache.

Software Multithreaded Transactions (SMTX) reproduces these simulated parallelizations on real hardware [13]. Since synchronization arrays do not exist in real hardware, Liberty Queues provide a high-performance software-only replacement. SMTX uses queues to communicate speculative loads and stores to a commit thread. For each speculative load or store, SMTX enqueues the address and value. Thus, the bandwidth demands on the queue can be twice as high as main memory bandwidth. This is a much more ambitious performance goal than previous core-to-core communication systems. By contrast, FastForward merely attempts to match the bandwidth of Gigabit Ethernet [2, 3]. In SMTX, the producer thread is always busier than the consumer thread, therefore Liberty Queues shift the communication burden to the consumer thread.

Liberty Queues achieve high bandwidth core-to-core communication on a wide range of multicore and multiprocessor systems. This paper shows experimental results on seven systems, including desktop and server microprocessors. The maximum core-to-core average bandwidth was 2.49 GB/s on a 2.4 GHz Xeon E5530, with an overhead of 3.21 ns per produce-consume pair. The minimum core-to-core average bandwidth was 281 MB/s on an Itanium 2, with an overhead of 15.9 ns. These performance results compare favorably with FastForward queues which achieved 281 MB/s with 28.5-31 ns overhead on a 2.66 GHz Opteron 2218 [3].

The remainder of this paper is organized as follows. Section 2 provides background. Section 3 describes the implementation of Liberty Queues. Section 4 describes how the highly optimized and architecture specific Liberty Queues were ported to Itanium. Section 5 presents a detailed evaluation on seven systems, including an Opteron, several Xeon of varying micro-architecture, two Itanium and a desktop Core 2 Duo, demonstrating overall performance stability and robustness. Section 7 discusses the problem of deadlock as it applies to lockless queues in pipeline-parallel programs. Section 8 discusses related work, and section 9 concludes.

2. Background

Multithreaded queue implementations based on locks are impractical. Diagnosing naïve queue implementations reveals two forms of unnecessary overhead: cyclic dependencies between producer and consumer and excess cache state transitions. There exist few efficient implementations for a single-producer/single-consumer (SP/SC) concurrent lock-free (CLF) queue: Lamport's CLF queue [6], the FastForward queue [3], Delayed-Buffer Lazy-Synchronization (DBLS) Queues [17], and MCRingBuffer [7, 8]. Other implementations of queues exist but focus on the related problem of a multiple-producer/multiple-consumer (MP/MC) CLF queue. MP/MC queues yield worse performance than the SP/SC case.

2.1 Lamport's CLF Queue

Lamport devised the first implementation of a CLF queue and proved that explicit synchronization is unnecessary under sequential consistency. Lamport's CLF queue does not require the use of a lock to serialize producer and consumer. In Lamport's queues use a circular buffer. A tail points to the first writable element in the buffer and a head points to the first readable element. The producer and consumer increment the tail or head pointer after writing or reading respectively. The consumer and producer check both that the head and tail pointers are not equal to avoid passing each other. Sharing control data between producer and consumer causes cache line thrashing.

2.2 The FastForward CLF Queue

The FastForward CLF queue improves on Lamport's implementation by optimizing cache behavior. in FastForward queues the producer and consumer have exclusive read and write access to the tail and head pointers, respectively. To avoid passing each other, the consumer writes NULL values after reading and the producer checks that array elements are NULL before overwriting them. The head and tail indexes are never shared and are always cache resident. To prevent cache line thrashing, FastForward temporally slips the producer and consumer. No enqueue and dequeue memory accesses operate on the same cache line.

2.3 DBLS and MCRingBuffer

DBLS and MCRingBuffer improve performance by employing batch updates to the head and tail pointers to reduce cache line contention. The producer and consumer will update private copies of the tail and head for several iterations before updating a shared copy. The differences between DBLS and MCRingBuffer are subtle and will be discussed in a later section. MCRingBuffer is the most similar implementation to Liberty Queues, so a more detailed description appears in the next section.

3. Implementation

This section describes the Liberty Queue's implementation, and contrasts this implementation with MCRingBuffer [7, 8].

3.1 Data Structures

Lamport's CLF queue causes cache thrashing on multicore multiprocessor systems. Each produce must read the tail pointer and update the head pointer. Each consume reads the head pointer and updates the tail pointer. Thus, for each produce or consume, a cache line on the modifying processor transitions from shared to modified[11]. Liberty Queues use a variation of the MCRingBuffer system of batch updates and cache-aware data layout to minimize cache thrashing.

Figure 1(a) shows the MCRingBuffer data structure, and Figure 1(b) shows the Liberty Queue data structure. Both data structures have three sections. The first section, contains the producer's local variables. These

```
/* Producer's local variables */
int localRead;
int nextWrite;
PAD(1, sizeof(uint64_t));

/* Control variables */
volatile int read;
volatile int write;
char cachePad2[CACHE_LINE - sizeof(int)];

/* Consumer's local variables */
int localWrite;
int nextRead;
char cachePad3[CACHE_LINE - sizeof(int)];

uint64_t *element;

```

(a) The MCRingBuffer internal data-structure.

```
/* Producer's local variables */
long nextWrite;
PAD(1, sizeof(uint64_t));

/* Control variables */
volatile int read;
volatile int write;
char cachePad2[CACHE_LINE - sizeof(int)];

/* Consumer's local variables */
int localWrite;
int nextRead;
char cachePad3[CACHE_LINE - 2 * sizeof(int)];

uint64_t *element;

```

(b) The Liberty Queue internal data-structure.

Figure 1. Comparison between Data Structures

variables are read and written by the producer and never the consumer. The next section contains control or global variables read and written by producers and consumers. The last section contains the consumer's local variables, which are read and written only by the consumer.

MCRingBuffer maintains a symmetrical relationship between producers and consumers. In the common fast case, producers and consumers may only update their local variables. In the rare case, consumers and producers update the control variables after consuming or producing a "batch" of data. Updating the shared control variables will yield cache ping-ponging, but the MCRingBuffer implementation only does this once per batch. Generally, large batch sizes yield fewer ping-pongs and better performance.

MCRingBuffer is a general-purpose communication framework. Liberty Queues, however, were designed around SMTX's workloads. For SMTX, the most common form of communication is from a process performing speculative computations to a process checking for misspeculation. The producer should always be “busier” than the consumer, and so Liberty Queues shift the burden of communication from the producer to the consumer. Whereas in MCRingBuffer, the control variables were owned equally by producer and consumer, in Liberty Queues, the producer enjoys an implicit “right of way” when accessing control variables.

The Liberty Queue’s producer no longer maintains a local copy of the read variable and reads the control variables directly. Additionally, the Liberty Queue’s `nextWrite` variable is 64-bit. The reason for this change will be explained in Section 3.2.

3.2 Producers

Figure 2(b) shows the Liberty Queue produce function. To produce a value, the Liberty Queue first writes a value to the queue. If this produce completes a batch of data, check to see if the producer is in danger of overtaking the consumer. If so, execute a callback and sleep, waiting for the situation to improve. If not, update the control variables to make writes visible to the consumer. Updating the control variables to make writes visible to the consumer is called flushing the queues and is analogous to flushing a file. For correctness, the callback function must flush the queue which called it. Further discussion of the callback’s purpose is deferred to Section 6.

Figure 2(a) shows the MCRingBuffer produce function. In contrast to the similarities in data structures between Liberty Queues and MCRingBuffer, the produce functions are very different. MCRingBuffer is non-blocking. Non-blocking queues are more general than their blocking counterparts. However, non-blocking queue implementations are vulnerable to deadlock if more than one queue is used. The problem of deadlock for queues is revisited in a later section, but it motivates the use of callbacks in Liberty Queue’s producer and consumer.

MCRingBuffer checks that the queue is not full before a produce operation. Liberty Queues check that there is room in the queue for the next produce, after producing a value unconditionally. This design decision is very unorthodox, and the motivation is to help compilers optimize the queues. Liberty Queues are intended to be inlined aggressively. From the compiler’s perspective, the sleep waiting loop may never halt. A compiler will not move the store to the queue above the check. Placing the checks after the store makes the store unconditional. This allows the compiler to schedule the store earlier for better instruction-level parallelism.

Liberty Queues utilize Streaming SIMD Extension’s (SSE) streaming store instruction for better bandwidth and performance stability. Streaming stores bypass L2 cache as soon as an entire cache line is written and violate x86’s usual coherence guarantees. Loads will not see the values of streaming stores until `sfence` instruction executes. The performance implications of this design decision appear in section 5.

```
function produce(uint64_t element)
{
  int afterNextWrite = NEXT(nextWrite);
  if(afterNextWrite == localRead) {
    if(afterNextWrite == read) {
      return INSERTED_FAILED;
    }
    localRead = read;
  }
  buffer[nextWrite] = element;
  nextWrite = afterNextWrite;
  wBatch++;
  if(wBatch >= batchSize) {
    write = nextWrite;
    wBatch = 0;
  }
  return INSERT_SUCCESS;
}
```

(a) The MCRingBuffer Produce Process

```
function produce(uint64_t value, function callBack)
{
  int index = nextWrite >> 32;
  uint64_t *data = truncate32(localWrite);
  streamWrite(data + index, value);
  nextWrite = NEXT(nextWrite);
  index = nextWrite >> 32;
  if(index % BATCH_SIZE == 0) {
    if(distance(nextWrite, read) < DANGER) {
      /* Blocking path */
      callBack();
      while(distance(nextWrite, read) < DANGER)
        usleep(10);
    } else {
      /* Fast path */
      memoryFence();
      write = index;
    }
  }
}
```

(b) The Liberty Queue Produce Process

Figure 2. Comparison between Produce Process

In the fast common case, MCRingBuffer uses variables `nextWrite`, `localRead`, `buffer`, `wBatch`, and `batchSize`. Liberty Queues only use `localWrite`. If the queue implementation is not inlined, MCRingBuffer will require five loads and three stores. By contrast, the non-inlined version of Liberty Queues require only one load and two stores per invocation. Liberty Queues sacrifice readability and flexibility for raw performance. In particular, Liberty Queues make batch size a compile time decision and require it to be a power of two for best performance. Additionally, Liberty Queues always allocate the underlying queue buffer somewhere in the lower 32 bits of memory. The lower 32-bits of `nextWrite` hold the buffer's address, and the upper 32-

```
function consume(uint64_t *element)
{
  if(nextRead == localWrite) {
    if(nextRead == write) {
      return EXTRACT_FAILED;
    }
    localWrite = write;
  }
  *element = buffer[nextRead];
  nextRead = NEXT(nextRead);
  rBatch++;
  if(rBatch >= batchSize) {
    read = nextRead;
    rBatch = 0;
  }
  return EXTRACT_SUCCESS;
}
```

(a) The MCRingBuffer Consume Process

```
uint64_t sq_consume(function callBack)
{
  if(nextRead == localWrite) {
    if(nextRead == write) {
      /* Blocking path */
      callback();
      while(write == nextRead)
        usleep(10);
    } else {
      /* Fast path */
      read = localRead;
    }
    localWrite = write;
  }
  uint64_t val = buffer[nextRead];
  nextRead = NEXT(nextRead);
  prefetch(buffer + nextRead + QPREFETCH);
  return val;
}
```

(b) The Liberty Queue Consume Process

Figure 3. Comparison between Consume Process

bits hold the index into the buffer. When inlining, GCC is clever enough to promote `nextWrite` to a register. Inlined Liberty Queues require zero loads and one store per produce.

3.3 Consumers

Figure 3(b) shows the Liberty Queue consume function. When consuming, the Liberty Queue first checks the local copy of the write variable to determine if any values are available to dequeue. If not, the queue consults the shared control variables. If this check fails as well, the callback function executes and the queue sleeps, waiting for the producer to enqueue more data. If the control variable indicates there are more values already enqueued, the consumer updates the read control variable with its copy of local state. Whether or not the queue blocked, the consumer updates the local copy of the write state. Copying the local read state to the read control variable is called reverse flushing. A correct callback function for the consumer must at least reverse flush the queue. At this point, regardless of the path of execution, there is at least one value ready to dequeue. The consumer dequeues a value and updates local state. Finally, the consumer prefetches a distant future value from the queue, and returns the dequeued value.

The consumer is simpler than the producer code, and more similar to the corresponding code in MCRing-Buffer, shown in Figure 3(a). The most important difference is that in Liberty Queues the consumer does not update the global read state unless it updates its local copy of the write state. This policy greatly favors producers over consumers. Every time the consumer writes to the control variables it will cause a cache ping-pong, so the consumer delays these writes as long as possible.

The consumer prefetches many accesses ahead using the 64-bit SSE prefetching mechanism. Unlike prefetching with a load instruction, the SSE prefetch instruction will never cause a stall, does not require a register, and will not fault if the address is unreadable. Prefectching improves bandwidth and performance stability as will be seen in the Experiments Section.

3.4 Tunable Parameters

Liberty Queues use three major tunable parameters: queue size, batch size, and prefetch distance. Larger queue size helps compensate for bursty communication patterns or consumers with variable processing times. Experiments showed that the only disadvantage of long queues was potentially exhausting 32-bit address space. Once the queues are long enough to buffer out bursts by producers and slow processing by consumers, there are no performance gains for increasing the size further. Liberty Queues default to 16 megabytes. Batch size determines how often the producer will update global state. Larger batches mean fewer updates and better performance, but updating too infrequently causes the consumer to spin wait uselessly. Batch size defaults to 128 kilobytes. Finally prefetch distance determines how early the consumer will read data. The idea is to request data far enough in advance that it reaches the L2 cache before a real request for that data. However, reading data too early can cause cache pollution. Liberty queues prefetch 1 kilobyte ahead.

The present implementation "bakes-in" all three tunable parameters at compile-time. As previously mentioned this avoids loads and register pressure at runtime.

3.5 128-bit Queues

Liberty Queues also have a 128-bit queue implementation. SMTX, Liberty Queue's most important client, enqueues a 64-bit address and a 64-bit value at the same time. The 128-bit queues reduce the overhead of producing very slightly by checking for queue fullness once for each 128-bits produced. On most platforms, 128-bit queues have very slightly better bandwidth than 64-bit queues.

4. Porting to EPIC

Porting to Itanium proved remarkably easy. First, the x86 specific inline assembly was replaced with Itanium equivalents. All of the changes were one-to-one replacements. x86-64 uses the exotic `movntiq` SSE instruction to execute streaming stores bypassing the cache and storing directly to memory. On Itanium the more mundane `st8.nta` instruction suffices. The `sfence` instruction used to ensure all streaming writes are visible before updating control state becomes IA-64's `mf` instruction. Similarly, the x86-64 non-temporal prefetch instruction `prefetch.nta` becomes `lfetch.nta` on IA-64.

At the outset of the project inline assembly was believed to limit Liberty Queues' portability, particularly since the queues rely on fairly obscure instructions. Since all of the instructions have direct equivalents on IA-64, there may be value in a thin portability library. Exploring the MIPS and ARM architectures reveal that they have the same primitives as x86-64 and IA-64 and could easily be targeted by future work.

On the other hand, porting from x86-64 Linux to IA-64 Linux proved relatively challenging contrary to initial expectations. x86-64 Linux provides a flag for the mmap syscall named MAP_32BIT, which can be used to force an anonymous mapping into the lower 32-bits of address space. Recall that the data field of the Liberty Queues must be in the lower 32-bits of memory. IA-64 Linux provides no comparable functionality. Even more frustratingly, requests for anonymous pages tend to be mapped immediately above the first 32-bits of memory. The long term solution to this problem will be to hack the IA-64 Linux kernel to support the MAP_32BIT flag. The short term solution is to mmap all queues into a region of memory with a fixed upper 32-bit address. The upper 32-bits become a constant chosen at compile time and used at runtime to reconstruct the full 32-bit buffer pointer from only the lower 32-bits.

5. Experiment Results

In this section, we performed experiments on seven different machines showing the Liberty queue yields a superior performance results.

5.1 Experimental Platforms

Machine Name	Module Number	Microarchitecture	Frequency	L2 Cache	CPU/ System	Die/ CPU	Core/ Die	Thread/ Core	Total Threads
Clovertown	Intel Xeon E5310	Clovertown	1.60 GHz	4 MB	2	2	2	1	8
Dunnington	Intel Xeon X7460	Dunnington	2.66 GHz	3 MB L2 16 MB L3	4	3	2	1	24
Conroe	Intel Core 2 Duo	Conroe	2.66 GHz	4 MB	1	1	2	1	2
Egypt	AMD Opteron Processor 846	Egypt	2.00 GHz	8 MB	2	1	2	1	4
Nehalem	Intel Xeon E5530	Nehalem	2.40 GHz	8 MB	1	1	4	2	8
McKinleyA	Intel Itanium 2	McKinley	1.5 GHz	256 KB	192	1	1	1	192
McKinleyB	Intel Itanium 2	McKinley	900 MHz	256 KB	2	1	1	1	2

Table 1. Experimental Platforms

Table 1 shows the experimental platforms used in this study. The Core 2 Duo is the only desktop-class system in the study. The Opteron system is a proxy for the Opteron 2218 system used to validate the FastForward queues. The Opteron 846 used in this study is from the generation immediately preceding the 2218, uses DDR instead of DDR2 memory, is clocked 660 MHz slower, and has half the L2 cache. The four- and six-core Xeon E5310 and X7460 are both relatively recent Xeon multicore multiprocessor systems and demonstrate how Liberty Queues perform when communicating across the systems' buses. The E5530 shows the affects of Nehalem's new memory architecture and hyperthreads. The two Itanium 2 systems show the performance of Liberty Queues on EPIC architectures.

5.2 Benchmarking

The benchmark program allocates two queues and forks. Let the parent process be called the producer and the child process the consumer.

The parent process calls `clock_gettime` and stores the current time. Then it produces one billion 64-bit integers to the first queue, while simultaneously computing their sum. The producer then consumes a 64-bit integer from the second queue. The producer prints out the sum and the value consumed. Finally, the parent process calls `clock_gettime` and prints the difference in time between the first and second calls.

The consumer process consumes one billion 64-bit integers from the first queue and computes their sum. After computing their sum, the consumer produces the value to the second queue and exits.

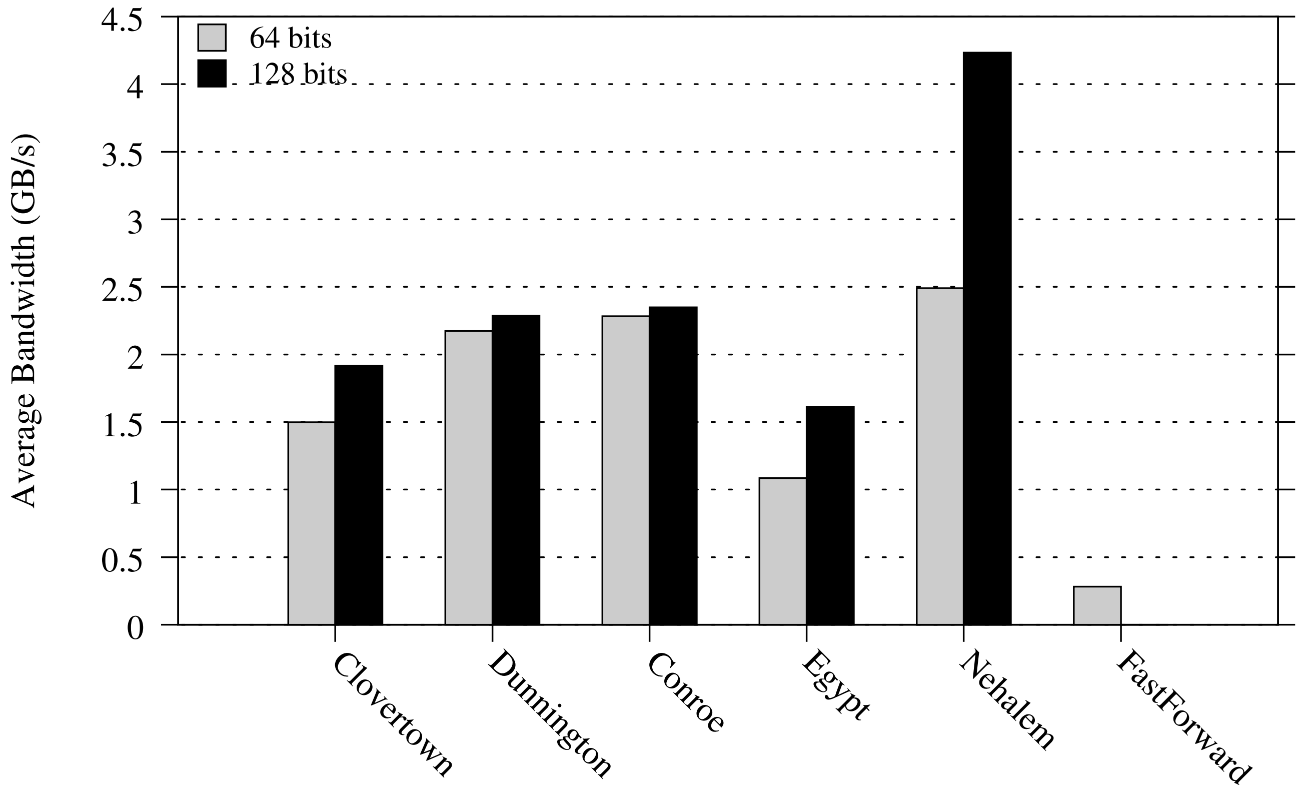


Figure 4. The bandwidth of 64- and 128-bit queues on each of the test systems and FastForward's reported bandwidth.

The overhead of the queues is the number of values produced and consumed divided by the running time of the benchmark as reported by the parent process. Each datapoint is the average of 16 runs of the benchmark per target.

5.3 Bandwidth

Figure 4 shows the bandwidth of Liberty Queues on each test system. The bandwidth of the original FastForward queue implementation is shown at the far right for comparison. The most surprising result is the performance of 128-bit produces on Nehalem. The origin of this speedup is still unknown. Nehalem may be fusing two 64-bit stores into a single 128-bit macro-operation. By contrast, the Itanium based systems both show a large slow down for 128-bit queues. A later subsection will reveal that code bloat is the source of this performance problem.

The initial MCRingBuffer evaluation[7, 8] achieved peak bandwidth of 1.19 GB/s for 64-bit produces between two cores on the same die and 430 MB/s between cores on different processors. The MCRingBuffer evaluation used a Xeon X5355, which is the same processor generation as *Clovertown*, but is clocked 1 GHz

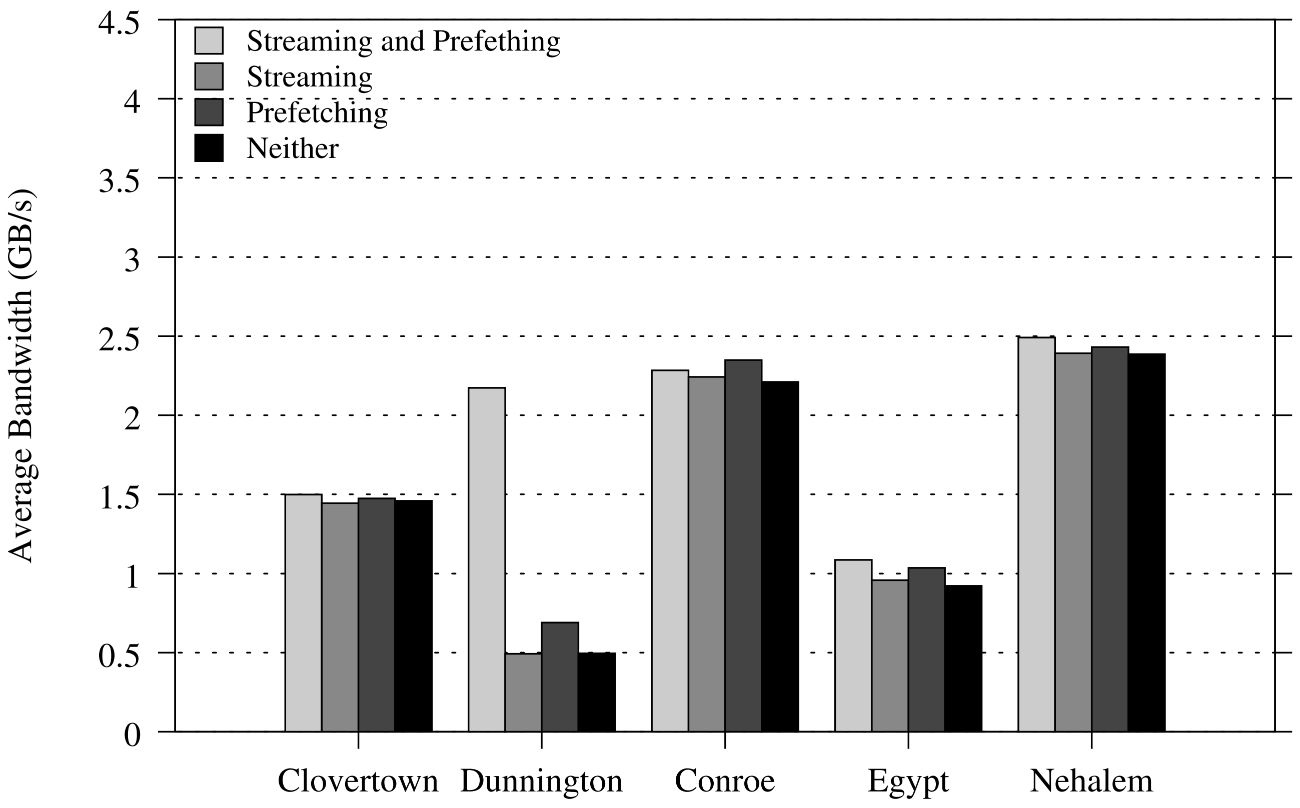


Figure 5. The bandwidth of 64-bit queues with prefetching and streaming enabled and disabled.

faster and features a faster FSB. *Clovertown*'s bandwidth with Liberty Queues is 1.53 GB/s for cores sharing an L2 cache, and 1.54 GB/s for unrelated cores.

5.4 Streaming

Figure 5 shows the performance consequences of streaming and prefetching. Streaming and prefetching are slightly advantageous on their own and in combination. The only exception is the *Dunnington* system. Performance counters indicate that the other systems automatically do stride prefetching. For Liberty Queues, *Dunnington* does not stride prefetch. Additionally, *Dunnington*'s L3 cache is too large. The other systems have smaller caches and must write to memory. *Dunnington*'s L3 cache is the same size as the queues, enqueued data stays on the producers processor. Performance-wise, it is more desirable for the queue data to go directly to main memory, where the consuming processor can find it. The streaming writes bypass the L3 cache, side-stepping the issue. Increasing the size of the queues by a factor of four and enabling prefetching yields similar performance to the prefetching-streaming implementation, but is obviously impractical.

Figure 6 shows the range of runtimes as a percentage of average runtime for the same data used in figure 5. Although streaming and prefetching has little affect on bandwidth on most applications, it does improve

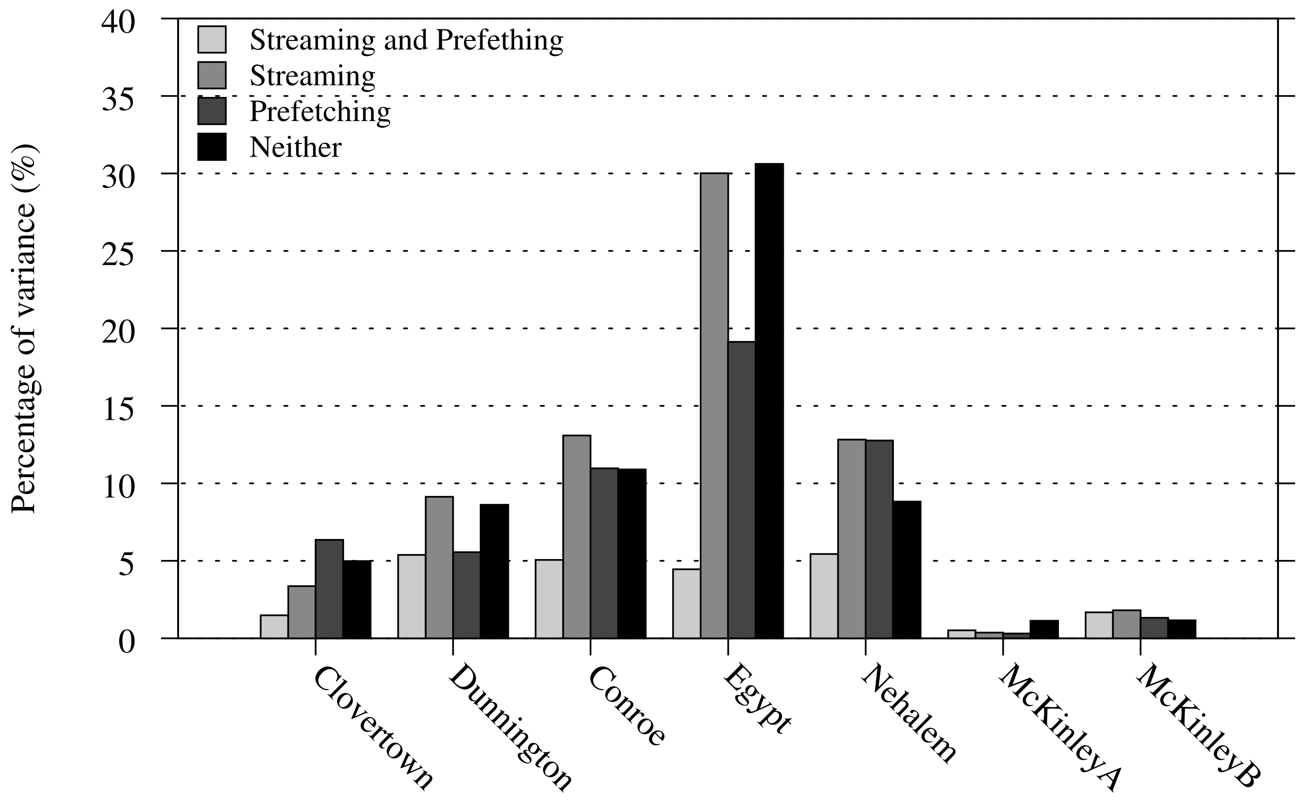


Figure 6. Range of runtimes as a percent of average runtime.

the consistency of results. The streaming-prefetching queues are never as fast as the fastest non-streaming non-prefetching queue runs, but also never as slow as the slowest. The next section will show that streaming and prefetching can either help or hurt performacne depending on the topological relationship between the producing and consuming cores. In this section, the operating system schedules producing and consuming processes without pinning by the user.

5.5 Inlining

The FastFoward queue implementation calls the produce and consume function through an indirect function pointer. In the interests of fair comparison, Figure 7 shows the bandwidth of Liberty Queues when compiled with `-fno-inline` and `-fPIC`. The bandwidth on all x86-64 test systems is greatly reduced, but the bandwidth on Itanium systems actually increased for 128-bit produces. This indicates that the inlined queues caused too much code bloat and overwhelmed the Itanium's instruction cache. On x86-64 systems, removing inlining increases the difference in performance between the 64- and 128-bit queues. This indicates that enqueue and dequeue operations are fast enough that function call overhead is non-trivial. Even without inlining, all Liberty Queue implementations outperform FastForward queues.

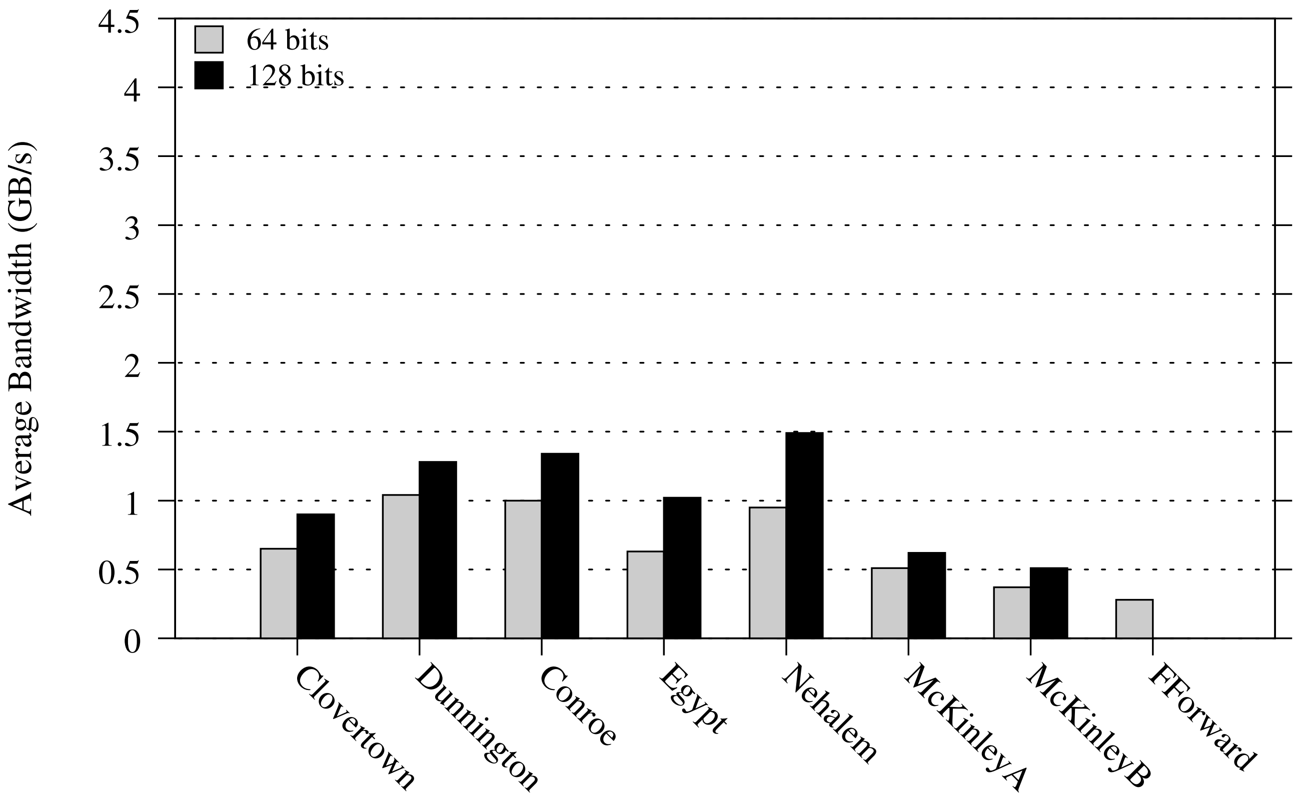


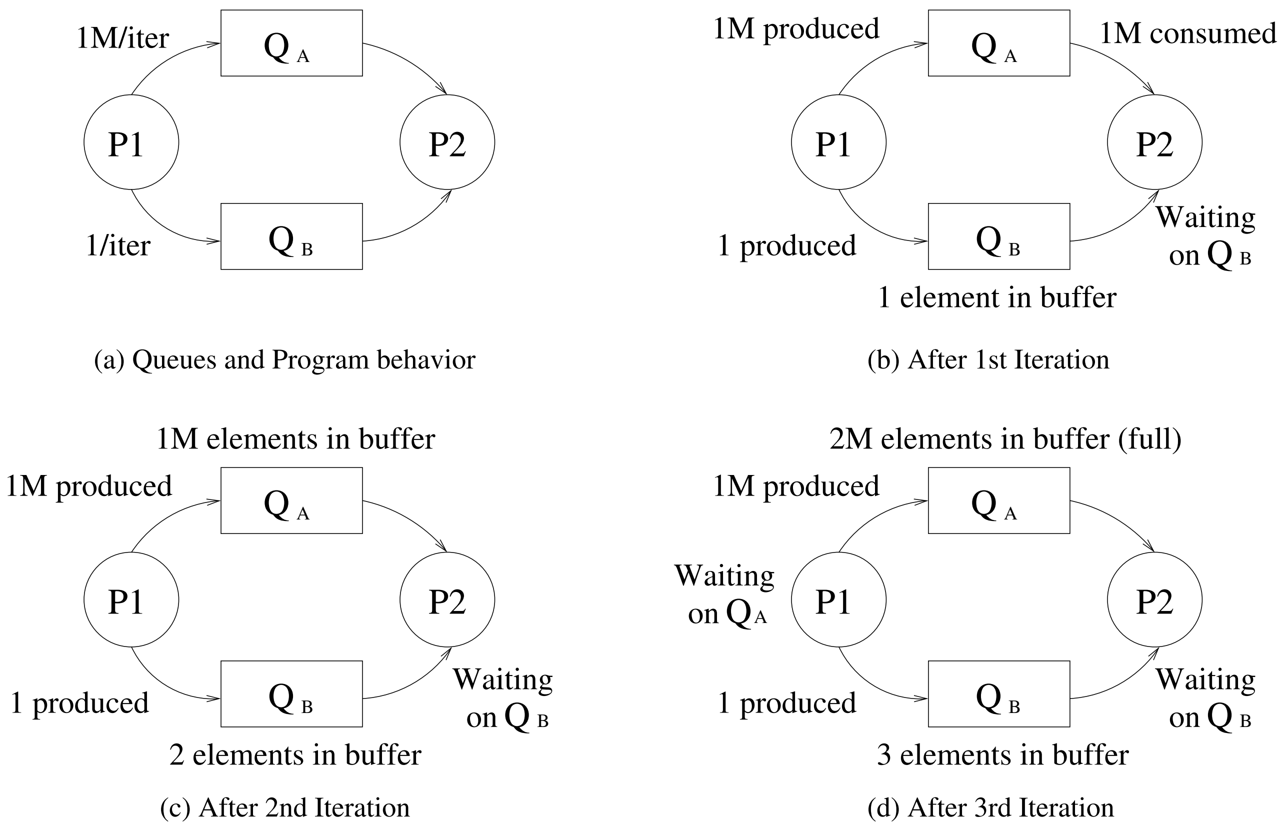
Figure 7. The bandwidth of 64- and 128-bit queues with position independent code and no inlining.

6. Deadlock

This section explains the call back functions first mentioned in Section 3.2. MCRingBuffers [7, 8] will suffer deadlock in some programs. Consider a program with two processes as shown in Figure 8 (a). In each iteration of the first process($P1$) produces one million words to the first queue(Q_A) and one word to the second queue(Q_B). The second process($P2$) will consume one million words from the first queue(Q_A) and one word from the second queue(Q_B) per iteration. This seemingly innocuous program will deadlock.

Without loss of generality, assume a chunk size of 16 kilowords and a queue size of 2 megawords. Q_A sends 32 batches per iteration, but Q_B sends one batch per 16,000 iterations. In the first iteration in Figure 8 (b), $P1$ produces one million 64-bit words with 32 batches to Q_A, and $P2$ consumes them. $P1$ produces one word to Q_B, but $P2$ waits on Q_B since P1 has not yet sent a batch.

In the second iteration in Figure 8 (c), $P1$ produces one million words, but $P2$ does not consume the words since it is still waiting for Q_B. At the end of the second iteration there are two words in Q_B buffer, one million words in Q_B, and $P2$ continues to wait on Q_B.



(a) Queues and Program behavior (b) After 1st Iteration (c) After 2nd Iteration (d) After 3rd Iteration

Figure 8. Deadlock Example

In the third iteration in Figure 8 (d), $P1$ produces one million words to Q_A, and one word to Q_B. At the end of the third iteration, there are two million words in Q_A completely filling it. $P2$ continues on Q_B.

Finally, in the fourth iteration P_1 cannot produce to Q_A since it is completely full. P_2 cannot read from Q_B since P_1 has not yet produced a full batch of data. The system is deadlocked.

To prevent deadlock, the Liberty Queue has a callback function, which is called right before the queue sleeps. The programmer or compiler using the queue will supply a callback function that flushes all outgoing queues and reverse flushes all incoming queues. By adding flush operations for other queues into the callback function, Liberty Queues avoid deadlock. In the previous example, when $P1$ sleeps due to Q_A, Q_B is flushed by callback execution. Now, $P2$ can consume data from Q_B, and data in Q_A will be consumed in the next iteration. Therefore, $P1$ and $P2$ avoid deadlock.

7. Related Work

Since many applications for multiprocessor systems use queue data structures, there exists a wide array of previous work [3, 5–10, 12, 15, 16] on enhancing queue performance. Initially, different lock implementations were tried in an attempt to reduced lock-based overhead (i.e. convoying and memory hot spots). These techniques minimized the overhead of blocking and contention points. Subsequently, lock-based implementations were abandoned in favor of non-blocking alternatives. Non-blocking implementations exposed more concurrency by not enforcing mutual exclusion.

Many non-blocking queue implementations [5, 10, 15, 16] focus on MP/MC CLF queues. These queues require higher overhead to avoid ABA problems [9]. Known solutions to the ABA problem rely on expensive hardware synchronization primitives (Compare-and-set, Compare-and-swap, or load-linked/store-conditional) and extra state information to maintain linearizability [4].

The three most closely related works to Liberty Queues are FastForward, MCRingBuffer, and DBLS. FastForward Queues are the least similar. Liberty Queues, FastForward, and MCRingBuffer all employ the same basic strategy. They avoid cache line thrashing by rarely reading or writing to global state. By contrast, FastForward has no global state. FastForward's producer and consumer know it is safe to write or read by checking for NULL or not NULL respectively.

The differences between MCRingBuffer and DBLS are very subtle. MCRingBuffer improves on DBLS by using cache-line padding to separate producer from consumer variables. DBLS updates cached state at fixed intervals. MCRingBuffer does not update cached state if the producer or consumer can continue to make progress with the current cached state.

Liberty Queues differs from MCRingBuffer, and DBLS by using aggressive prefetching and streaming. Additionally, Liberty Queues favor producer performance over consumer performance. MCRingBuffer and DBLS have no such bias.

8. Conclusion

Liberty Queues provide high performance core-to-core communciation for IA-64 and x86-64 architectures. Liberty Queues achieve 500 MB/s on a first generation Itanium 2. Compared with FastForward's 281 MB/s bandwidth on a modern Opteron, and MCRingBuffer's 430 MB/s bandwidth on modern Xeons. When executing on systems more closely mirroring the MCRingBuffer and FastForward test systems, Liberty Queues achieve bandwidths of 1.09 GB/s and 1.50 GB/s respectively. Nehalem offers the best performance at 2.49 GB/s.

Liberty Queues implementation most closely resembles MCRingBuffer, but is much more carefully tuned. Additionally, MCRingBuffer treats producers and consumers equally, but Liberty Queues shifts most of the overhead to the consumer end of the queue. This customization is useful in situations where the producer is expected to be busier than the consumer. The opposite customization favoring the consuming process over the producing process may be useful for other parallelization systems.

Porting Liberty Queues from x86-64 to IA-64 proved comparatively painless despite the highly optimized and architecture sensitive nature of the queues. A second generation Liberty Queue implementation for IA-64 could improve substantially on the performance in this work. A compiler level implementation of Liberty Queues would be able to schedule producer code into unused instruction slots, allowing produces to execute for free.

The paper evaluates Liberty Queues on seven different systems with two different architectures. We explain the performance variations and experimentally justify the Liberty Queue's design decisions. Liberty Queue's high bandwidth and low overhead laid the foundations for the SMTX parallelization framework

References

[1] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. doi: http://dx.doi.org/10.1109/MICRO.2007.35.

[2] J. Giacomoni and M. Vachharajani. *Efficient Point-To-Point Enqueue and Dequeue Communications*. US Patent Application. November 2008.

[3] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 43–52, New York, NY, USA, February 2008. ISBN 978-1-59593-795-7. doi: http://doi.acm.org/10.1145/1345206.1345215.

[4] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/78969.78972.

[5] E. Ladan-mozes and N. Shavit. An optimistic approach to lock-free fifo queues. In *In Proceedings of the 18th International Symposium on Distributed Computing, LNCS 3274*, pages 117–131. Springer, 2004.

[6] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, April 1983. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/69624.357207.

[7] P. P. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *ANCS'09: ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2009.

[8] P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *IPDPS '10: Proceedings of the 24th International Parallel and Distributed*

Processing Symposium, April 2010.

[9] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998. ISSN 0743-7315. doi: http://dx.doi.org/10.1006/jpdc.1998.1446.

[10] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 253–262, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: http://doi.acm.org/10.1145/1073970.1074013.

[11] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, New York, NY, USA, 1984. ACM. ISBN 0-8186-0538-3. doi: http://doi.acm.org/10.1145/800015.808204.

[12] S. Prakash, Y. H. Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Trans. Comput.*, 43(5):548–559, 1994. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.280802.

[13] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative Parallelization Using Software Multi-threaded Transactions. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010.

[14] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[15] W. N. Scherer, III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147–156, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: http://doi.acm.org/10.1145/1122971.1122994.

[16] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 134–143, New York, NY, USA, 2001. ACM. ISBN 1-58113-409-6. doi: http://doi.acm.org/10.1145/378580.378611.

[17] C. Wang, H.-S. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: http://dx.doi.org/10.1109/CGO.2007.7.