Noname manuscript No.
(will be inserted by the editor)

B-Queue: Efficient and Practical Queuing for Fast Core-to-Core Communication

Junchang Wang · Kai Zhang · Xinan Tang · Bei Hua

Received: date / Accepted: date

Abstract Core-to-core communication is critical to the effective use of multi-core processors. A number of software based concurrent lock-free queues have been proposed to address this problem. Existing solutions, however, suffer from performance degradation in real testbeds, or rely on auxiliary hardware or software timers to handle the deadlock problem when batching is used, making those solutions good in theory but difficult to use in practice. This paper describes the pros and cons of existing concurrent lock-free queues in both dummy and real testbeds and proposes B-Queue, an efficient and practical single-producer-single-consumer concurrent lock-free queue that solves the deadlock problem gracefully by introducing a self-adaptive backtracking mechanism. Experiments show that in real massively-parallel applications, B-Queue is faster than FastForward and MCRingBuffer, the two state-of-the-art concurrent lock-free queues, by up to 10x and 5x, respectively. Moreover, B-Queue outperforms FastForward and MCRingBuffer in terms of stability and scalability, making it a good candidate for fast core-to-core communication on multi-core architectures.

Keywords nonblocking synchronization · lock-free queue · parallelization · multi-core · backtracking

Junchang Wang · Kai Zhang
University of Science and Technology of China (USTC), Hefei 230027, China
Suzhou Institute for Advanced Study, USTC, Suzhou 215123, China

Xinan Tang
Intel Compiler Lab, Intel Corporation, Santa Clara, California 95054, USA

Bei Hua
University of Science and Technology of China (USTC), Hefei 230027, China
Suzhou Institute for Advanced Study, USTC, Suzhou 215123, China
Tel.: 86-551-3607043
Fax: 86-551-3607043
E-mail: bhua@ustc.edu.cn

1 Introduction

As multi-core architectures become ubiquitous, great efforts have been made to parallelize single-threaded applications [4, 19, 28, 34, 35]. One example is that a great deal of work applies pipeline parallelism to sequential network applications [7, 34, 35]. Harnessing abundant CPU resources, however, is still a big challenge. One problem is the lack of a fast and efficient core-to-core communication mechanism on existing commodity multi-core platforms, and threads have to rely on shared memory to exchange information [6]. Take 10Gbps network as an example, a minimal 64-byte Ethernet frame must be processed in amortized time of 67 nanoseconds, which is about one DRAM access time. If there is no efficient core-to-core communication, parallelizing such network applications cannot take effect because a single memory based core-to-core communication might offset the benefit from parallelization.

A large body of work focuses on providing core-to-core communication with First-In-First-Out (FIFO) queues [7, 12, 15, 18, 19, 27, 35]. It is generally accepted that a lock-based solution is inappropriate for applications featuring fine-grained parallelism, and a lock-free design is a promising approach. Unfortunately, general purpose lock-free FIFOs still do not perform adequately. For example, *cache-unaware* lock-free FIFOs [14, 16] take more than 1,000 CPU cycles to insert or extract an element. To tackle this problem, FastForward [7] and subsequent research efforts [18,35] provide single-producer-single-consumer concurrent lock-free FIFOs (abbreviated as CLF queues from here on) to support fast core-to-core communication; all of them try to avoid cache thrashing as much as possible. Experiments on dummy testbeds [7,18] show that these solutions take 20 cycles on worst-case for each insert/extract operation, making CLF queues a promising candidate for fast core-to-core communication on multi-core architectures.

When employing existing CLF queues in practice, we encountered the following problems. (1) Peak performance in dummy testbed sometimes is misleading, and in real applications performance of these CLF queues decreases dramatically. Experimentally we found that the claimed peak performance is achieved only when CLF queues stores in L1 cache (Up to 97% memory accesses must be served by L1 cache), and no cache thrashing occurs. However, in real applications, L1 cache miss and cache thrashing are unavoidable [20, 32]. (2) Performance of existing CLF queues drops dramatically when the number of queues increases. This performance degradation has been observed and reported in [18, 35], but not carefully studied in the literature. (3) Existing CLF queues are hard to use. FastForward [7] relies on predefined thresholds to avoid cache thrashing, but these thresholds vary from system to system and are hard to tune in practice. Other work [12, 18, 35] relies on batching to achieve maximum performance. However, to avoid the deadlock problem inherent in batching (Section 2.5), auxiliary timer and threads must be used to periodically check the state of a CLF queue to keep the consumer alive. The auxiliary timer and threads disturb CLF queue's cache behavior, and complicate the system as well (Section 2.6).

This paper studies CLF queues in building up a real multi-10Gbps network processing system where parallelism is widely used [6, 7, 18, 35], and both the hardware and software capabilities are stressed [5, 8]. The major contributions of this paper are as follows:

– Existing CLF queues are evaluated on both dummy and real applications; their strengths on dummy testbeds and weaknesses on real applications are studied. The evaluation comes to the conclusion that for CLF queues, no further effort should be made to pursue higher performance on dummy testbeds. In contrast, great attention must be paid to putting them into practice.
– A fast yet practical CLF queue, named B-Queue, is proposed. B-Queue outperforms existing CLF queues in terms of scalability and stability and requires no parameter tuning. All these advantages come from a novel backtracking mechanism that can adaptively adjust consumption to production without the need of any auxiliary mechanism or manually tuned parameters.

The remainder of this paper proceeds as follows. Section 2 provides the background and motivation of this paper. Section 3 describes B-Queue's design, and proves its correctness in Section 4. Section 5 evaluates the performance of CLF queues on dummy and real testbeds. Section 6 discusses related work and Section 7 concludes.

2 Backgrounds and Motivation

Three parallel programming techniques — *task parallelism*, *data parallelism*, and *pipeline parallelism* — are widely used in parallelizing real world applications. *Task parallelism* uses multiple independent and often heterogenous tasks and is usually used for relatively long duration tasks. *Data parallelism* applies the same computation to independent data elements in parallel. These two techniques, however, fail to parallelize applications that have strict ordering requirements in computation [7]. Two examples are network processing and stream processing where there exist a partial or total order in computation, making them poor candidates for task- and data-parallel techniques [6].

Pipeline parallelism, on the other hand, is applicable to applications which feature a total order on computation tasks. In *pipeline parallelism*, a single task is divided into several pipeline stages each of which operates concurrently. A large amount of work has been done to exploit pipeline parallelism in real world applications [25, 29, 30], and a 3-stage-n-way pipeline model [6, 7, 18, 35] is widely used in parallelizing network applications. High performance CLF queues are extensively studied as a mechanism to pass data from between consecutive pipeline stages. We survey the existing work on CLF queues in the following subsections.

2.1 Lock Based Queues

The simplest way to implement a shared queue is to use locks. However, lock based queues are inefficient because both the producer and consumer need to acquire a lock before accessing the queue, which prevents concurrent access to the queue even if different slots are accessed.

2.2 Lamport's CLF Queue

```
L01: BOOL enqueue( ELEMENT_TYPE value )
L02: {
L03:   if (NEXT(head) == tail){
L04:     return FAILURE;
L05:   }
L06:   buffer[head] = value;
L07:   head = NEXT(head);
L08:   return SUCCESS;
L09: }

L10: BOOL dequeue( ELEMENT_TYPE *value )
L11: {
L12:   if (head == tail) {
L13:     return FAILURE;
L14:   }
L15:   *value = buffer[tail];
L16:   tail = NEXT(tail);
L17:   return SUCCESS;
L18: }
```

Fig. 1: Lamport's queue

Lamport presented the first CLF queue in [17], where he proved that under sequential consistency memory model, locks could be removed from single-producer-single-consumer queues, resulting in lock-free queues. Figure 1 gives the pseudo-code of Lamport's CLF queue, where the exclusion of explicit synchronization allows the producer and consumer to concurrently access the queue.

However, since the producer and consumer use two shared variables, *head* and *tail*, for implicit synchronization, Lamport's CLF queue suffers from *cache thrashing*. The cache line containing *head* and *tail* is frequently invalidated by the modification of the two control variables, causing the cache line bouncing back and forth between two caches. In addition to that, Lamport's CLF queue cannot be used in architectures with weak memory consistency models, such as PowerPC and IA64 [7].

2.3 FastForward

FastForward improves Lamport's CLF queue by eliminating shared variables between producer and consumer. Figure 2 shows the pseudo-code of FastForward where *head* and *tail* become non-shared local variables (The *head* is a local variable of the producer and the *tail* is a local variable of the consumer). Moreover, coupling makes FastForward execute correctly even on processors with weak memory consistency [7]. It is worth noting that cache thrashing still occurs if two buffer slots indexed by *head* and *tail* are located in the same cache line. To avoid cache thrashing, FastForward introduces a *temporal slipping* mechanism (Figure 3) to ensure that the producer and consumer are separated by a certain *distance*.

The first problem of FastForward is that it requires heavy work to manually tune parameters to achieve peak performance, and thus lacks stability in practice. The

```
F01: queue_init()
F02: {
F03:   buffer[0..end] = NULL;
F04: }

F05: BOOL enqueue( ELEMENT_TYPE value )
F06: {
F07:   if (NULL != buffer[head]) {
F08:     return FAILURE;
F09:   }
F10:   buffer[head] = value;
F11:   head = NEXT(head);
F12:   return SUCCESS;
F13: }

F14: BOOL dequeue( ELEMENT_TYPE *value )
F15: {
F16:   *value = buffer[tail];
F17:   if (NULL == value) {
F18:     return FAILURE;
F19:   }
F20:   buffer[tail] = NULL;
F21:   tail = NEXT(tail);
F22:   return SUCCESS;
F23: }
```

Fig. 2: FastForward implementation

```
A01: adjust_slip() {
A02:   dist = distance(producer, consumer);
A03:   if (dist < DANGER) {
A04:     dist_old = 0;
A05:     do {
A06:       dist_old = dist;
A07:       spin_wait(avg_time * ((GOOD+1)-dist));
A08:       dist = distance(producer, consumer);
A09:     } while (dist < GOOD && dist_old < dist);
A10:   }
A11: }
```

Fig. 3: Suggested slip adjustment routine in FastForward

efficiency of *temporal slipping* heavily relies on two pre-defined thresholds, *GOOD* and *DANGER*, and Fastforward suggests that *GOOD* and *DANGER* should be the size of 6 cache lines and 2 cache lines, respectively. In experiments, however, we found that the optimal values vary from system to system, and the suggested values only fit one of the three servers used in our experiments. For each system, we have to measure FastForward for many times to get optimal values for *GOOD* and *DANGER* before applying it in practice.

One technical merit of FastForward is that when *head* and *tail* pointers are far enough apart cache misses are avoided. However, the adjustment routing itself, shown in Figure 3, touches the indexes of both producer and consumer (Line A02 and A08) to calculate the distance between *head* and *tail* pointers, inevitably incurring cache thrashing because one of these two variables is modified by another thread. Therefore

the adjustment routine cannot be invoked frequently, leaving to the users a question of how often to call the adjustment routine.

2.4 Multi-line Updates

Noticing the cache thrashing issue, multi-cache-line update is proposed in [35]. The idea is that the producer does not update *head* until enough data has been accumulated to fill in one or a few cache lines. However, such aggressive batching makes the queue prone to deadlock in real applications. We discuss the problem in next subsection because MCRingBuffer suffers the same problem.

```
M01: BOOL Insert(T element) {
M02:   afterNextWrite = NEXT(nextWrite);
M03:   if (afterNextWrite == localRead) {
M04:     if (afterNextWrite == head) {
M05:       return INSERT_FAILED;
M06:     }
M07:     localRead = head;
M08:   }
M09:   buffer[nextWrite] = element;
M10:   nextWrite = afterNextWrite;
M11:   wBatch ++;
M12:   if (wBatch >= batchSize) {
M13:     tail = nextWrite;
M14:     wBatch = 0;
M15:   }
M16:   return INSERT_SUCCESS;
M17: }

M18: BOOL Extract(T* element) {
M19:   if (nextRead == localWrite) {
M20:     if (nextRead == tail) {
M21:       return EXTRACT_FAILED;
M22:     }
M23:     localWrite = tail;
M24:   }
M25:   *element = buffer[nextRead];
M26:   nextRead = NEXT(nextRead);
M27:   rBatch++;
M28:   if (rBatch >= batchSize) {
M29:     head = nextRead;
M30:     rBatch = 0;
M31:   }
M32:   return EXTRACT_SUCCESS;
M33: }
```

Fig. 4: MCRingBuffer implementation

2.5 MCRingBuffer

To make the algorithm cache-aware, MCRingBuffer [18], shown in Figure 4, adopts several optimization techniques, including *Cache-line Protection* and *Batch Update of Control Variables*. The idea is to use a local *head* (Line M03) to shadow the global

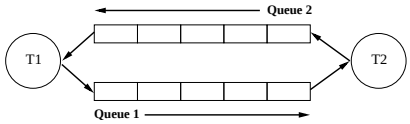


Fig. 5: Deadlock example

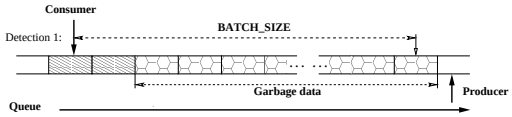


Fig. 6: Deadlock Prevention in MCRingBuffer

head, and most of the time only local variable is accessed to avoid cache thrashing (Line M09).

Batching improves performance but makes the algorithm prone to deadlock. For example, when processes are working with multiple MCRingBuffer queues and they form a circular chain, deadlock occurs when each process is waiting for others to finish. An example is shown in Figure 5, where thread *T1* generates and passes data to thread *T2* through *Queue 1*, and thread *T2* generates and passes data to thread *T1* through *Queue 2*. This scheme is widely used to synchronize two threads where *T1* passes data to *T2* and gets feedbacks from *T2*. When *T1* generates a few data that is less than the size of *batchSize*, and polls *Queue 2* trying to get feedbacks, and at the same time *T2* generates some feedbacks that are less than the size of *batchSize*, and polls *Queue 1* trying to get more data, deadlock occurs.

2.6 Deadlock Prevention

Some deadlock prevention methods have been proposed in [12, 18]. The basic idea is to periodically inject garbage data into the queue to keep the consumer alive. For example, MCRingBuffer [18] suggests that the producer periodically injects unused elements, and then the consumer discards them (Figure 6). However, none of these methods in the literature has been implemented and measured. In this subsection, we propose some common mechanisms for deadlock prevention and outline the difficulties inherent in each of the proposals. After that, we come to the conclusion that preventing deadlock is non-trivial and the proposals fail to achieve this goal.

(1) A naive way to prevent deadlock is to use a timer and an auxiliary monitoring thread. This thread periodically checks whether there is a deadlock, and injects garbage data whenever necessary. However, the extra deadlock-check thread fundamentally complicates synchronization, and changes a single-producer-single-consumer queue into a multi-producer-single-consumer queue. To the best of our knowledge, no multi-producer-single-consumer queues in the literature can provide fast core-to-core communication required by fine-grained parallelism.

(2) The second way is to use a hardware/software timer that periodically interrupts the producer to inject garbage data. However, as the producer's execution path may

be interrupted, producer must be reentrant. To the best of our knowledge, none of the existing CLF queues is both lock-free and reentrant.

(3) The third way also uses a hardware/software timer, but the timer periodically informs the producer by writing a global variable instead of interrupting the producer. Producer actively checks this global variable to see whether the queue needs a flush operation. Unfortunately, this method cannot solve the deadlock problem shown in Figure 5. Think about the scenario where *T1* tries to get data from *T2* by polling *Queue 2*, and *T2* tries to get data from *T1* by polling *Queue 1*. If neither queue has enough data, both threads are busy waiting and have no chance to check the global variable. Deadlock remains unsolved.

(4) A new method, discussed in [12], improves method (3) by adding a callback function that flushes all outgoing queues. The callback function is called right before the queue sleeps or polls a specific queue. For example, in Figure 5, when thread *T1* fails to get a new datum from *Queue 2*, the callback function is executed to flush *Queue 1* by, for example, inserting garbage data. A detailed analysis of this method, however, shows that implementing this method is a non-trivial work, especially for systems that can dynamically create and destroy CLF queues. Actually, this method works by binding all the queues in the system, which increases system complexity. Whenever a thread fails to get a new datum, the program must decide whether to flush the remaining queues, and which queues should be flushed. To make a decision, the system must maintain a global data structure that propagates the status of each queue in the system. Therefore, this method only fits special use cases because it shifts the responsibility of deadlock prevention to users.

To sum up, none of the above deadlock prevention methods is practical. Even worse, they generally cause significant performance degradation. For example, MCRingBuffer with deadlock prevention method (3) suffers performance degradation by up to 5 times (Section 5.3).

3 B-Queue

The analysis in section 2 shows that existing CLF queues are good in theory but difficult to use in practice. This section presents the design of B-Queue and how *backtracking* gracefully solves the deadlock problem.

3.1 Batching

Figure 7 shows the pseudo-code of *enqueue* and *dequeue* operations. Two local control variables, *head* and *tail*, are used to record current positions of producer and consumer, respectively. Another two local control variables, *batch_head* and *batch_tail*, are used by the producer and consumer to probe a group of available slots. Slots between *head* and *batch_head* are safe for the producer to store data, and slots between *tail* and *batch_tail* are safe for the consumer to read.

For the producer, the *head* and *batch_head* are initialized to zero. When the producer wants to insert an element, it first compares the *head* with *batch_head* to see

```
Q01: BOOL enqueue( ELEMENT_TYPE value )
Q02: {
Q03:   if (head == batch_head) {
Q04:     if (buffer[MOD(head+BATCH_SIZE)])
Q05:       return FAILURE;
Q06:     batch_head=MOD(head+BATCH_SIZE);
Q07:   }
Q08:   buffer[head] = value;
Q09:   head = NEXT(head);
Q10:   return SUCCESS;
Q11: }

Q12: BOOL dequeue( ELEMENT_TYPE *value )
Q13: {
Q14:   if (tail == batch_tail) {
Q15:     if (backtrack_deq() != SUCCESS)
Q16:       return FAILURE;
Q17:   }
Q18:   *value = buffer[tail];
Q19:   buffer[tail] = NULL;
Q20:   tail = NEXT( tail );
Q21:   return SUCCESS;
Q22: }
```

Fig. 7: B-Queue Algorithm

if there are any empty slots available (Line Q03). If no *empty* slot is available, it probes the slot that is *BATCH_SIZE* slots ahead of current position to see if a block of *BATCH_SIZE empty* slots could be found (Line Q03-Q06). The *MOD* is a modular operation against the queue size. The producer returns if no enough *empty* slots are available; otherwise, the *batch_head* is updated and an element is inserted into the queue (Line Q08-Q10). As long as the *head* does not catch up with the *batch_head*, the producer only executes the fast path (Line Q08-Q10) to insert elements.

For the consumer, it first compares the *tail* with *batch_tail* to check if there are any filled slots available (Line Q14). If no filled slot is available, it probes a block of *filled* slots using *backtracking* (discussed in section 3.2) (Line Q15). It returns *FAILURE* if no *filled* slot is available; otherwise, it gets an element from the queue, clears the slot, and updates variable *tail* (Lines Q18-Q20). As long as the *tail* does not catch up with the *batch_tail*, the consumer only executes the fast path (Line Q18-Q21) to extract elements.

The basic idea behind batching is that both the producer and consumer detect a batch of available slots at a time, ideally reducing the number of shared memory accesses by *(BATCH_SIZE-1)*/ *BATCH_SIZE*. If *BATCH_SIZE* is set properly, the producer and consumer will never operate on the same cache line. In addition to cache thrashing avoidance, batching also facilitates hardware prefetching that may greatly improve the performance of CLF queues. Experiments show that on Intel processors with hardware prefetching features on, as high as 99% of cache accesses are served by L1 data cache in dummy testbeds (see Section 5.2).

B-Queue is easy to use in real applications. *BATCH_SIZE* can be simply set to the size of multiple cache-lines, and performance of B-Queue is insensitive to *BATCH_SIZE* (Section 5.4). This is different from the system-dependent, pre-defined thresholds in FastForward.

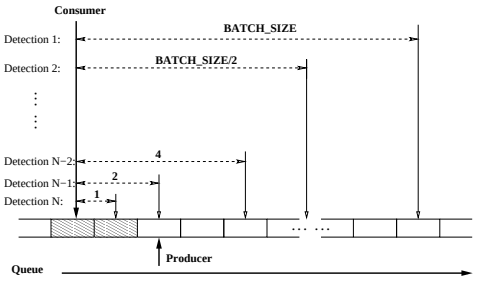


Fig. 8: Searching for a groups of filled slots

```
B01: BOOL backtrack_deq( )
B02: {
B03:   batch_size = BATCH_SIZE;
B04:   batch_tail = MOD(tail+batch_size-1);

B05:   while (!buffer[batch_tail]) {
B06:     spin_wait(TICKS);
B07:     if (batch_size > 1) {
B08:       batch_size = batch_size >> 1;
B09:       batch_tail = MOD(tail+batch_size-1);
B10:     }
B11:     else
B12:       return FAILURE;
B13:   }
B14:   return SUCCESS;
B15: }
```

Fig. 9: Backtracking to search for filled slots

3.2 Backtracking

A detailed analysis of deadlock prevention mechanisms commonly used in existing CLF queues has been presented in Section 2.6. In B-Queue, a novel deadlock prevention mechanism, *backtracking*, is designed for *consumer* to adaptively find filled slots if *producer* halts temporarily. Figure 8 depicts the *backtracking* mechanism, and Figure 9 presents its pseudo-code. The batching size variable, *batch_size*, is initialized to *BATCH_SIZE* (Line B03), and the *batch_tail* is *BATCH_SIZE* ahead of *tail* (Line B04). In each run, the consumer checks the status of *buffer[batch_tail]*, where *batch_tail* is the edge of this batching. If the slot is not filled (Line B05), batching size is halved (Line B08) and *batch_tail* is recalculated (Line B09). This process repeats until *buffer[batch_tail]* is found filled or the batching size reaches zero. In the former case, a block of *filled* slots are found and *batch_tail* is updated; in the later case, no data is available and *FAILURE* is returned. It is worth noting that in both cases, *backtracking* completes within finite time, a key element in our proof of correctness (Section 4.3).

Take Figure 8 as an example. The producer inserts two elements and then halts. The consumer first checks *buffer[tail + BATCH_SIZE-1]* (*Detection 1*). Because the slot is not filled, the consumer automatically decreases the batching size and checks

```
N01: BOOL backtrack_deq( )
N02: {
N03:   if (batch_history < BATCH_MAX) {
N04:     batch_history = MIN(BATCH_MAX, \
                         batch_history + INCREMENT);
N05:   }
N06:   batch_size = batch_history;
N07:   batch_tail = MOD(tail+batch_size-1);

N08:   while (!buffer[batch_tail]) {
N09:     spin_wait( TICKS );
N10:     if( batch_size > 1 ) {
N11:       batch_size = batch_size >> 1;
N12:       batch_tail = MOD(tail+batch_size-1);
N13:     }
N14:     else
N15:       return FAILURE;
N16:   }
N17:   batch_history = batch_size;
N18:   return SUCCESS;
N19: }
```

Fig. 10: Backtracking with automatic adjustment

buffer[tail + BATCH_SIZE/2-1] (*Detection 2*). This process repeats until *buffer[tail+1]* is found filled (*Detection N*).

Backtracking prevents deadlock by removing the necessary condition of deadlock, *circular wait*. Instead of letting the producer and consumer wait for each other to finish, *backtracking* allows the consumer to actively decrease its batching size. Binary search algorithm (Line B08) is used to quickly approach the *filled* slots. In the worst case, *backtracking* takes $log_2^{BATCH\_SIZE}$ memory accesses to find the first hit.

To summarize, *backtracking* is a simple yet efficient deadlock prevention mechanism. It gracefully solves the deadlock problem by adaptively decreasing batching size according to producer's speed at runtime. It is simple because no timer or auxiliary monitoring thread is required, and thus system complexity is not added. It is efficient because no garbage data is generated.

3.3 Self-Adaptive Backtracking

Batching increases latency [18, 35]. Although *backtracking* solves deadlock by automatically adjusting batching size, B-Queue still suffers high latency when the producer is not busy. For example, when there is only one filled slot in the array and the producer halts, the time that the consumer takes to find the filled slot could be:

$$log_2^{BATCH\_SIZE} * TICKS; \quad (1)$$

If BATCH_SIZE is 512 and TICKS equals to 2,000 CPU cycles, the latency is as high as 18,000 CPU cycles ($log_2^{512} * 2000$).

To tackle this problem, a *self-adaptive backtracking* is presented in Figure 10. The constant BATCH_SIZE is replaced by a global variable *batch_history* that records the history of *batch_size*. After a successful run of *backtrack_deq()*, the value of

batch_size is stored in *batch_history* (Line N17), and used as the start value of *batch_size* in next run (Line N04). In this way, when the producer is not busy, the start value of *batch_size* decreases, so does the latency. However, when the producer gathers speed, the value of *batch_size* should be enlarged. A possible solution is that whenever *backtrack_deq()* is invoked and *batch_history* is less than *BATCH_MAX*, *batch_history* increases by *INCREMENT* (Line N03-N05), where *INCREMENT* could be the size of a cache line to avoid cache thrashing.

The biggest strength of *self-adaptive backtracking* is that it can adaptively adjust the batching size and make a trade-off between latency and performance.

4 Correctness

To prove the correctness of B-Queue, we recall that our queue is based on a statically allocated memory (*buffer*). By correctness of B-Queue, we mean that the consumer dequeues elements in the same order that they were enqueued by the producer. Due to lack of space, we regard the executions of the producer and consumer are consistent with their program orders, respectively. Besides, it is also reasonable to assume that (1) aligned word-sized accesses (both read and write) are atomic, and that (2) the *buffer* has a low bound of 1 and a conceptually infinite high bound because the index of the position wraps around the buffer if needed with assistance from function MOD() and NEXT() in B-Queue. These assumptions are justified on modern multi-core processors. We will call all these assumptions M.

4.1 Safety

B-Queue is safe because it satisfies the following properties:

(1) All the elements that have been inserted by the producer and have not been extracted by the consumer compose an array (denoted as L and ranges from index tail to head), because once an element is inserted (index l in L), its next position $l+1$ will be the position to be inserted into, and the element indexed by $l+1$ cannot be extracted until the element indexed by l has been extracted by the consumer.

(2) Elements are only inserted into the most significant position (referenced by head) in array L, because index *head* is a private variable of the producer, and for each enqueue operation, the producer places an element in the position referenced by *head* and then monotonically advances *head* by one.

(3) Elements are only extracted from the less significant position (referenced by tail) in array L, because index $tail$ is a private variable of the consumer, and for each dequeue operation, the consumer extracts elements from position referenced by $tail$ and then monotonically advances $tail$ by one.

Initially, all the properties hold. By induction (Section 3.1) we could see that they continue to hold when B-Queue makes progress and we have the following theorem.

Theorem 1 *In B-Queue, the consumer dequeues elements in the same order that they were enqueued by the producer.*

4.2 Linearizability

B-Queue has no critical section, but obviously there are *linearization points* [9, 10]. For enqueue() method, Line Q08 is a linearization point where the message indicating a successful enqueue operation is propagated. This linearization point takes effect when the new value has been successfully written into the queue [1]. If the queue is full, the enqueue() method has a linearization point where it return a failure (Line Q05). Similarly, for dequeue() method, Line Q16 and Q19 are linearization points. It is worth noting that linearization point Line Q19 takes effect after the value has been successfully read (Line Q18). Since the queue itself always reflects the state of *full* or *empty*, the queue never enters a transient state in which the state of the queue can be mistaken.

4.3 Liveness

B-Queue is *wait-free* because both the enqueue() and dequeue() operations are guaranteed to complete within finite time. This property guarantees that either the producer or the consumer that takes steps makes progress. For example, in the scenario where producer halts half-way through enqueuing a value *x*, then the consumer will either throw a *FAILURE* (Line Q16) if the producer halted before storing the item in an empty array, or it will return a value if the producer halted afterward. There is a while loop diagram in *backtracking* (Figure 9 and Figure 10) which sometimes is called by the consumer. This while loop, however, will either take finite steps to find a hit (Section 3.1), or after $log_2^{BATCH\_SIZE}$ memory accesses, return a *FAILURE* that makes the consumer return immediately. Similarly, in the scenario where consumer halts half-way through dequeuing a value, then the producer will either throw a *FAILURE* (Line Q05) if the consumer halted before getting the item from a full array, or it will store a value if the consumer halted afterward.

Notably, the wait-free property of B-Queue implies that B-Queue is lock-free.

5 Evaluation

This section compares the performance of three CLF queues: FastForward, MCRingBuffer and B-Queue. Our experiments show that in real massively-parallel applications with more than six queues,

- B-Queue is about 10 times faster than FastForward;
- B-Queue is 5 times faster than MCRingBuffer if the deadlock prevention mechanism is on;
- *Backtracking* adds less than one nanosecond of delay.

[1] In actually, writing value and propagating message take effect at the same time.

5.1 Experiment Setup

Three servers with different architectures are used in the experiments. The first server is equipped with one Intel L5640 Westmere hex-core processor running at 2.26GHz. Each core has a L1 cache of 64KB and a L2 cache of 256KB. A 12MB L3 cache is shared among all cores. The processor has an integrated memory controller that supports DDR3 1333MHz memory, and 8GB memory is installed.

The second server has two Intel E5620 Westmere quad-core processors running at 2.4GHz. Each E5620 processor has a shared L3 cache and an integrated memory controller that supports DDR3 1066MHz memory, with 4GB memory installed. Two E5620 processors are connected by a QuickPath Interconnect (QPI) [24] at 5.86GT/s. Cores within the same CPU die (called *sibling cores*) communicate through L3 cache, and *non-sibling cores* communicate through QPI.

The third server is equipped with two Intel E7310 quad-core processors running at 1.6GHz. Each E7310 is composed of two replicas of dual-core modules; each module has two cores and a shared 4MB L2 cache. Front Side Bus is used to connect processors with the memory controller that supports DDR2 667MHz memory with 4GB memory installed. Cores within the same module (*sibling cores*) communicate through L2 cache, and *non-sibling cores* communicate through FSB.

All the three servers run 64-bit Linux 2.6.39 kernel, and the CLF queues are compiled by GCC 4.5.1 with -O2 option. Without optimization options, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results [1]. Turning on optimizations (-O1 and higher) makes the compiler attempt to improve the performance by compiling all statements at once to a single output object file, and allow the compiler to adopt optimizations such as *guess-branch-probability* and *merge-constants* [1] to reduce code size and execution time. Experimentally, we found that with optimization flag -O1, -O2 or -O3, all the CLF queues (FastForward, MCRingBuffer and B-Queue) perform around three times faster than versions compiled without an optimization flag. To make a fair comparison, all the CLF queues are compiled with -O2 option in experiments because optimization flags belonging to -O3 level do not benefit CLF queues.

For the MCRingBuffer queue, we sent our implementation to the original author for code review and received confirmation that our implementation is identical to what he did in [18]. For the FastForward queue, we received an implementation from the original author. B-Queue has been released under the GPL licence [2]. For FastForward and MCRingBuffer, optimizations are applied as much as possible to make a fair comparison. Those include cache line protection [18] and warm-up of queues before operation [7]. Performance critical parameters of FastForward are tuned manually in each experiment to get the best performance.

In each run, the producer thread inserts one trillion elements into the CLF queue, and the consumer thread gets these elements in order. Each thread is bound to a dedicated core, with the producer and consumer running on different cores. Hardware based *Time Stamp Counter* [11] in X86 is used to record the value of RDTSC register before and after the one trillion enqueue() and dequeue() operations to calculate the

[2] http://sourceforge.net/p/bqueue/code

total CPU cycles. The time of a single enqueue() or dequeue() operation is calculated by dividing the total execution time with one trillion and then subtracting workloads. It is worth noting that reading RDTSC register is not a serializing instruction. The read operation neither waits for all the previous instructions to finish before reading the counter, nor prevents subsequent instructions from starting to execute before the read operation is performed [11]. Therefore, there are some deviations in the total number of CPU cycles. Nevertheless, the deviation of a single enqueue() or dequeue() operation is negligible, as the number of total CPU cycles is divided by one trillion. For other architectural statistics (such as cache miss numbers), OProfile [3] is used to collect hardware performance counters. Each experimental data is averaged on 30 trials.

5.2 Performance Evaluation on Dummy Testbed

In this subsection, we measure the performance of the three CLF queues on a dummy testbed, and analyze the necessary requirements for achieving peak performance. The queue size is set to contain 2,048 elements, and the batch size is 256 elements.

Table 1: Peak performance of CLF queues

CLF queue	E5620	E7310	L5640
FastForward	14	15	14
MCRingBuffer	12	12	12
B-Queue	12	12	12

(a) On die (CPU cycles)

CLF queue	E5620	E7310	L5640
FastForward	27	36	–
MCRingBuffer	15	31	–
B-Queue	13	31	–

(b) Cross die (CPU cycles)

Average cycles per operation for on-die communication (two threads reside on the same die) and cross-die communication (two threads reside on different dies) are listed in Table 1(a) and 1(b), respectively. Table 1(a) shows that in the best case (on-die communication), all the three queues achieve peak performance with each enqueue/dequeue operation taking a dozen of CPU cycles, and moreover the peak performance is platform independent. In the cross-die case [3], MCRingBuffer and B-Queue use batching to amortise the overhead induced by cross-die interconnection, and outperform FastForward in both servers. Performance of all the three queues decreases on E7310 because the cross-die communication stresses the FSB on E7310. New generation point-to-point interconnects (QPI [24] from Intel and HyperTransport [13] from AMD corporation) provide much higher bandwidth and lower latency.

[3] We did not experiment this on L5640 because this server has only one processor.

Table 2: Cache behaviour of queues on E5620

CLF queue	L1 hit	L2 hit	L3 hit	Sibling	System
FastForward	96.99%	0.32%	0.00%	2.69%	0.00%
MCRingBuffer	99.83%	0.00%	0.00%	0.17%	0.00%
B-Queue	99.89%	0.02%	0.00%	0.09%	0.00%

(a) On die (sibling cores)

CLF queue	L1 hit	L2 hit	L3 hit	Sibling	System
FastForward	97.97%	0.16%	0.16%	0.00%	1.70%
MCRingBuffer	99.90%	0.01%	0.00%	0.00%	0.09%
B-Queue	99.94%	0.02%	0.00%	0.00%	0.04%

(b) Cross die

In addition to that, new servers (E5620) uses larger CPU cache and higher memory frequency. Those features significantly benefit concurrent lock free applications running on multi-core servers.

OProfile analyzes the cache behavior of the three CLF queues with assistance of hardware performance counters including L1 data cache hits (*L1 hit*), L2 cache hits (*L2 hit*), L3 cache hits (*L3 hit*), number of misses served by sibling core's cache (*Sibling*) and the system (*System*). Table 2 presents the cache behavior of CLF queues on E5620. The cache behavior on other two machines has a similar trend. Table 2 shows that all of the three CLF queues have extremely high L1 hit rate, and up to 96.99% memory accesses are served by L1 cache. That explains why reported peak performance can only be achieved on dummy testbeds, where the working set is small and nearly all of the memory accesses can be served by L1 cache.

However, real applications (Section 5.3) usually have larger memory footprint or use multiple CLF queues at the same time. In these cases, the working set of a CLF queue could not be held in the L1 cache, thereby causing *capacity cache misses* and *coherency cache misses* [20, 32]. Take E5620 as an example, a L1 cache hit costs 2 CPU cycles, a L2 cache hit costs 10 CPU cycles, a L3 cache hit costs 40 CPU cycles, and a memory access costs up to 200 CPU cycles. Apparently, a L3 cache hit alone may prevent a CLF queue from achieving the claimed peak performance reported in the literature. In other words, a CLF queue that behaves well on dummy testbeds will not behave the same in real applications. We demonstrate this in next subsection.

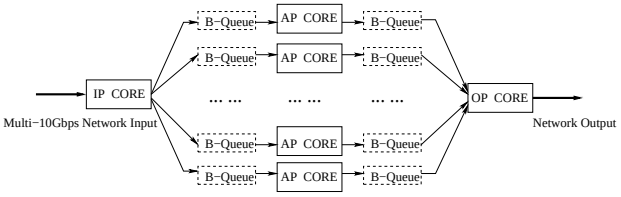
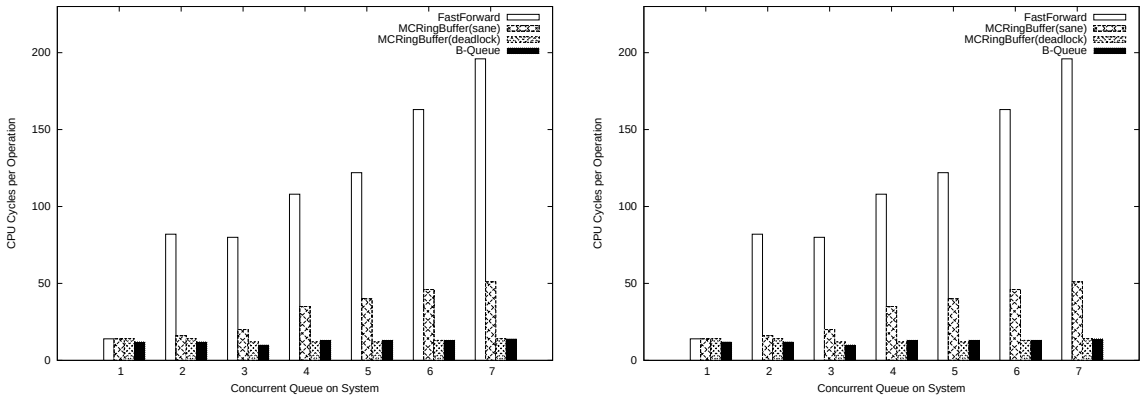


Fig. 11: Parallel multi-10Gbps networking system

5.3 Performance Evaluation on Real Testbed

In this subsection, we evaluate the three CLF queues in a real application. We use CLF queues to build a real multi-10Gbps network processing system that involves Layer 2 to Layer 7 (L2-L7) functions, including a TCP/IP stack from Libnids [2], a port-independent protocol identifier, and a HTTP parser that analyzes HTTP traffic. Figure 11 illustrates the pipelined organization of the system. There are three pipeline stages in the system:

- *Input Stage* (shown IP in Figure 11): One core in this stage receives packets from 10Gbps network interface cards (NIC) through an optimized Linux NIC driver [8], and then performs load balance by distributing the packets among multiple pipelines.
- *Application Stage* (shown AP in Figure 11): Each core in this stage gets packets by polling a CLF queue that connects it to the IP core. Then it performs a complete Layer 2 to Layer 7 network processing using the run-to-completion model, and sends the results to OP core through another CLF queue.
- *Output Stage* (shown OP in Figure 11): One core in this stage checks the CLF queues in a round-robin manner to collect the results.



(a) E5620 (b) E7310

Fig. 12: CLF queue performance in 1-to-N scenarios

Table 3: Cache behaviour with seven queues

	L1 hit	L2 hit	L3 hit	Sibling	System
FastForward	79.28%	8.50%	0.83%	5.32%	6.07%
MCRingBuffer	99.66%	0.00%	0.03%	0.07%	0.24%
B-Queue	99.79%	0.03%	0.01%	0.10%	0.07%

The queue size is set to contain 2,048 elements, and the batch size is 256 elements for MCRingBuffer and B-Queue. Figure 12(a) and 12(b) present the performance of different CLF queues in our parallel system built on E5620 and E7310, respectively. Since there are multiple CLF queues in the system, we enhance MCRingBuffer by adding the deadlock prevention mechanism (3) discussed in Section 2.6. To prevent an element from being permanently stalled in the queue, a timer and an auxiliary thread are added to periodically indicate the producer to inject garbage data, and the garbage data is discarded by the consumer. The original MCRingBuffer is denoted as *MCRingBuffer(deadlock)* in Figure 12, and the enhanced MCRingBuffer with deadlock prevention mechanism is denoted as *MCRingBuffer(sane)*. Performance of MCRingBuffer(sane) depends on the frequency on which garbage data is inserted; generally lower frequency results in better performance but longer latency. To deliver all of the 2,048 elements in the queue at peak rate without introducing extra latency, the auxiliary thread signals the producer every 10 microseconds ($\frac{(12cycles/datum)*2048}{2.4*10^9Hz}$) in E5620.

Both Figure 12(a) and Figure 12(b) show that as the number of CLF queues (AP cores) increases, performance of FastForward and MCRingBuffer(sane) decreases dramatically. Performance of FastForward starts to decrease when two queues are used. The reason is that the pre-defined thresholds only fit systems with one queue. When multiple queues are used, these pre-defined thresholds are no longer applicable, and cache thrashing occurs. For example, the L1 cache hit rate of FastForward drops from 96.99% in Table 2(a) to 79.28% in Table 3, and the cost of an enqueue() or dequeue() operation increases from 10+ CPU cycles to 150+ cycles.

Readers may notice that MCRingBuffer without deadlock prevention (denoted as MCRingBuffer(deadlock)) performs as well as B-Queue in Figure 12. The reasons are that (1) MCRingBuffer(deadlock) fortunately does not incur a deadlock in this testbed, and that (2) aggressive batching favors consecutive input in this experiment. We list MCRingBuffer(deadlock) to make a fair comparison. However, one should not use MCRingBuffer without deadlock prevention mechanism in practice (Section 2.5). Figure 12 shows that MCRingBuffer with a deadlock prevention mechanism is not efficient in practice. Performance of MCRingBuffer(sane) starts to decrease when more than three queues are used, and degrades 5 times when seven queues are used. The performance degradation is mainly ascribed to the garbage data and the deadlock prevention mechanism that introduces system overhead like context switches.

B-Queue aims at efficiency and usability. We design B-Queue with the notion that no running system dependent parameters should be involved and that no extra system complexity should be added. Figure 12 shows that the performance of B-Queue hardly decreases even if seven queues are used, and only 0.07% of memory accesses go to the main memory (Table 3). This experiment demonstrates that B-Queue is an efficient and robust core-to-core communication mechanism that may act as a building block for fine-grained parallelism.

5.4 Parameters in B-Queue

In this section, we evaluate how queue size, batch size, and workload affect the performance of B-Queue. We run the experiment on the E5620 server with one queue. As B-Queue is insensitive to the location of producer and consumer, we only give out the experimental data obtained in on-die case.

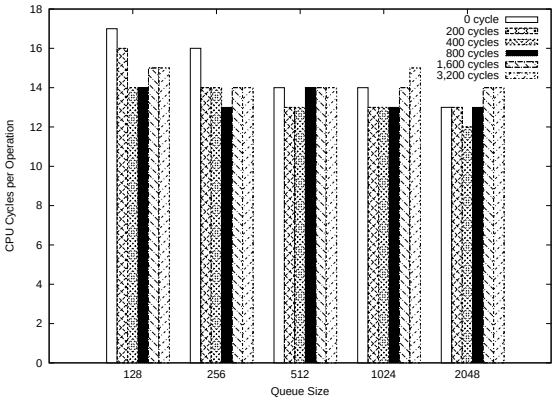


Fig. 13: Performance of B-Queue with different queue sizes and workloads

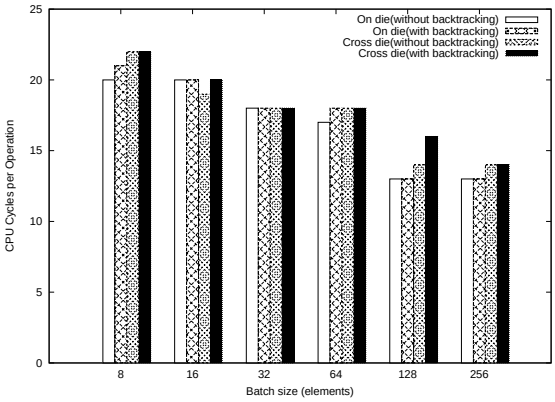


Fig. 14: Performance of B-Queue with backtracking

Figure 13 presents the average CPU cycles per operation (enqueue() or dequeue()). In each iteration, the consumer gets an element from the queue, and then waits for some given CPU cycles to simulate the workload. Five queue sizes (128, 256, 512, 1024 and 2048 elements) and six dummy workloads (0, 200, 400, 800, 1,600, and

3,200 CPU cycles) are used. It is clear that B-Queue is insensitive to both queue size and workload.

Figure 14 shows the average CPU cycles per operation that B-Queue (with and without *backtracking*) takes when different batch sizes are used. We use the evaluation method described in Section 5.2. Figure 14 shows that the performance of B-Queue increases with a larger batch size, and the maximum performance is achieved when batch size equals to 256. To evaluate the overhead of *backtracking*, we turn off *backtracking* and do the same experiments. Experimental data shows that *backtracking* adds less than two CPU cycles (that is one nanosecond on a 2GHz CPU) of overhead in both on-die and cross-die cases.

Experiments in Section 5.2, Section 5.3, and this section show that B-Queue is insensitive to *queue size*, *workload*, *number of concurrent threads*, and the *location of producer and consumer*. Moreover, the deadlock prevention mechanism, *backtracking*, barely introduces overhead.

In dummy testbeds, FastForward and MCRingBuffer have come to similar conclusions in respective papers. With dummy workload and a single CLF queue, FastForward is insensitive to workload, queue size and core allocation (Section 5.4 in [7]), and MCRingBuffer is almost insensitive with respect to queue size, core allocation and batch size (Evaluation 1 and 2 in [18]). In summary, all the three CLF queues (FastForward, MCRingBuffer and B-Queue) perform quite well in dummy testbeds with a single CLF queue, which is even comparable to hardware core-to-core communication mechanisms [19]. However, Section 5.3 shows that performance of FastForward and MCRingBuffer decrease dramatically in real applications. This set of experiments and analysis demonstrates that no further effort should be made to improve CLF queues on dummy testbeds. In contrast, the community should focus on problems in putting these CLF queues into practice.

6 Related Work

Since queues are widely used in multithreaded programs for communication, there are a wide array of studies on concurrent lock-free queues [14, 16, 21, 23, 33]. Most of these studies focus on multiple-producer and/or multiple-consumer queues. However, these queues often have limited performance, as a large portion of work has to be done to avoid ABA problem [22]. For example, a single enqueue operation may take more than one microsecond [14].

Being a special case, single-producer-single-consumer (SPSC) CLF queue is a promising candidate for high speed core-to-core communication [7, 12, 17, 18, 27, 31, 35], as it fundamentally avoids the ABA problem. So far, cache-aware SPSC CLF queues are studied and reported in [7, 18, 35], and the ILP optimization is reported in [12].

General purpose CLF queues commonly use linked list that requires dynamic memory management and may result in poor cache locality and extra synchronization [14, 16, 21, 31]. By exchanging data between producer and consumer through a statically allocated array, ring-buffer based queues [7, 18, 35] can exploit cache locality and facilitate the hardware cache prefetching.

Another approach is hardware queues [15, 19, 26] that provide instructions for enqueue and dequeue operations to reduce the overhead of software queues. However, hardware queues require to modify processors, and also impose challenges to the operating system as the queue state must be preserved across context switch. To date, hardware queues mainly exist in simulators, and none of the general purpose processors supports this feature yet.

7 Conclusion and Future Work

This paper presents B-Queue, a cache-aware CLF queue for fast core-to-core communication. Batch operation and backtracking are incorporated elegantly, where batching allows B-Queue to get high performance by avoiding cache thrashing, and backtracking prevents deadlock by adaptively adjust the batching distance according to the production speed. No running system parameter is used, and no system complexity is added. B-Queue improves the performance of existing CLF queues in terms of stability, scalability and it is simpler to use. The efficiency of B-Queue is demonstrated on a real testbed where multiple B-Queues are used.

Real applications featuring fine-grained parallelism require practical and efficient lock-free data structures. However, most lock-free data structures are very good in theory but difficult to use in practice and hard to tune for high-performance. All existing solutions including B-queue are only partial solutions to this important research topic. This is an open area that requires continuing attention.

Acknowledgements This work was supported by the "Fundamental Research Funds for the Central Universities (Grant No. WK0110000007)" from Chinese government. We would like to thank Andreas Voellmy from Yale University for helpful discussions on the subject. Also, the authors are grateful to anonymous reviewers whose valuable comments helped to improve the quality of this paper.

References

1. Gcc manual Http://http://gcc.gnu.org/onlinedocs/gcc-4.5.4/gcc/
2. Libnids Http://libnids.sourceforge.net/
3. Oprofile Http://oprofile.sourceforge.net/
4. Allen, M.D., Sridharan, S., Sohi, G.S.: Serialization sets: a dynamic dependence-based parallel execution model. In: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '09, pp. 85–96 (2009)
5. Dobrescu, M., Egi, N., Argyraki, K., Chun, B.G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., Ratnasamy, S.: Routebricks: exploiting parallelism to scale software routers. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, pp. 15–28 (2009)
6. Giacomoni, J., Bennett, J.K., Carzaniga, A., Sicker, D.C., Vachharajani, M., Wolf, A.L.: Frame shared memory: line-rate networking on commodity hardware. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems, ANCS '07, pp. 27–36 (2007)
7. Giacomoni, J., Moseley, T., Vachharajani, M.: Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPoPP '08, pp. 43–52 (2008)
8. Han, S., Jang, K., Park, K., Moon, S.: Packetshader: a gpu-accelerated software router. SIGCOMM '10, pp. 195–206 (2010)
9. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (March 2008)

10. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. pp. 463–492 (1990)
11. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 2B. 253669-033US (2009)
12. Jablin, T.B., Zhang, Y., Jablin, J.A., et al.: Liberty queues for epic architectures. In: Proceedings of the Eigth Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology, EPIC'10 (2010)
13. Keltcher C.N.; McGrath, K.J..A.: The amd opteron processor for multiprocessor servers. IEEE Micro (2003)
14. Kogan, A., Petrank, E.: Wait-free queues with multiple enqueuers and dequeuers. In: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP '11, pp. 223–234. New York, NY, USA
15. Kumar, S., Hughes, C.J., Nguyen, A.: Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In: Proceedings of the 34th annual international symposium on Computer architecture, ISCA '07, pp. 162–173 (2007)
16. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free fifo queues. Distributed Computing pp. 323–341 (2008)
17. Lamport, L.: Specifying concurrent program modules. ACM Trans. Program. Lang. Syst. pp. 190–222 (1983)
18. Lee, P., Bu, T., Chandranmenon, G.: A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In: 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS '10, pp. 1–12 (2010)
19. Lee, S., Tiwari, D., Solihin, Y., Tuck, J.: Haqu: Hardware accelerated queueing for fine-grained threading on a chip multiprocessor. In: The 17th IEEE International Symposium on High Performance Computer Architecture, HPCA-17 (2011)
20. Luan, H., Du, X.Y., Wang, S.: Prefetching j+-tree: A cache-optimized main memory database index structure. Journal of Computer Science and Technology pp. 687–707 (2009)
21. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, PODC '96, pp. 267–275
22. Michael, M.M., Scott, M.L.: Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING pp. 1–26 (1998)
23. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures, SPAA '05, pp. 253–262 (2005)
24. Molka, D., Hackenberg, D., Schone, R., Muller, M.S.: Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (2009)
25. Navarro, A., Asenjo, R., Tabik, S., Cascaval, C.: Analytical modeling of pipeline parallelism. In: 18th International Conference on Parallel Architectures and Compilation Techniques, PACT '09, pp. 281–290 (2009)
26. Ottoni, G., Rangan, R., Stoler, A., August, D.I.: Automatic thread extraction with decoupled software pipelining. In: In Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture, pp. 105–118. IEEE Computer Society (2005)
27. Preud'homme, T., Sopena, J., Thomas, G., Folliot, B.: Batchqueue: Fast and memory-thrifty core to core communication. In: 2010 22nd International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD '10, pp. 215 – 222 (2010)
28. Price, G.D., Giacomoni, J., Vachharajani, M.: Visualizing potential parallelism in sequential programs. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, PACT '08, pp. 82–90 (2008)
29. Sanchez, D., Lo, D., Yoo, R.M., Sugerman, J., Kozyrakis, C.: Dynamic fine-grain scheduling of pipeline parallelism. In: International Conference on Parallel Architectures and Compilation Techniques, PACT '11, pp. 22–32 (2011)
30. Thies, W., Chandrasekhar, V., Amarasinghe, S.: A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In: 40th IEEE/ACM International Symposium on Microarchitecture, MICRO '07, pp. 356–369 (2007)
31. Torquati, M.: Single-producer/single-consumer queues on shared cache multi-core systems. CoRR (2010)

32. Transier, F., Sanders, P.: Engineering basic algorithms of an in-memory text search engine. ACM Trans. Inf. Syst. pp. 2:1–2:37 (2010)
33. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, SPAA '01, pp. 134–143 (2001)
34. Upadhyaya, G., Pai, V.S., Midkiff, S.P.: Expressing and exploiting concurrency in networked applications in aspen. In: In Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 13–23 (2007)
35. Wang, J., Cheng, H., Hua, B., Tang, X.: Practice of parallelizing network applications on multi-core architectures. In: Proceedings of the 23rd international conference on Supercomputing, ICS '09, pp. 204–213 (2009)