ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/213894711

FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue

Conference Paper · January 2008

DOI: 10.1145/1345206.1345215

CITATIONS
157

READS
1,431

3 authors, including:

Tipp Moseley
University of Colorado Boulder
22 PUBLICATIONS **1,021** CITATIONS
SEE PROFILE

Manish Vachharajani
F5 Networks
58 PUBLICATIONS **1,965** CITATIONS
SEE PROFILE

All content following this page was uploaded by Manish Vachharajani on 22 February 2015.

The user has requested enhancement of the downloaded file.

FastForward for Efficient Pipeline Parallelism

A Cache-Optimized Concurrent Lock-Free Queue

John Giacomoni
University of Colorado at Boulder
john.giacomoni@colorado.edu

Tipp Moseley
University of Colorado at Boulder
tipp.moseley@colorado.edu

Manish Vachharajani
University of Colorado at Boulder
manish.vachharajani@colorado.edu

Abstract

Low overhead core-to-core communication is critical for efficient pipeline-parallel software applications. This paper presents FastForward, a cache-optimized single-producer/single-consumer concurrent lock-free queue for pipeline parallelism on multicore architectures, with weak to strongly ordered consistency models. Enqueue and dequeue times on a 2.66 GHz Opteron 2218 based system are as low as 28.5 ns, up to 5x faster than the next best solution. FastForward's effectiveness is demonstrated for real applications by applying it to line-rate soft network processing on Gigabit Ethernet with general purpose commodity hardware.

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

General Terms Algorithms, Experimentation, Performance

Keywords FastForward, linearizability, lock-free, multicore, multiprocessors, nonblocking synchronization, pipeline parallel, queue

1. Introduction

Traditionally, increases in transistor count and fabrication technology have led to increased performance. However, this trend has slowed due to limitations arising from power consumption, design complexity, and wire delays. In response, designers have turned to multicore configurations that incorporate multiple cores on one or more dies. While multicore configurations are a boon to throughput driven applications such as web servers, single-threaded applications' performance is stagnant. Furthermore, the typical approaches used to parallelize software have difficulty with general purpose applications [2] as the strategies have been to find, extract, and run nearly independent code regions on separate processors [23].

Recent work shows that many inherently sequential applications can be parallelized by using pipeline parallelism [8, 19, 20, 23, 27], including network-processor applications [5]. In this paradigm, computation is divided into sequential stages where each stage is a thread and is isolated from interference by being bound to separate cores. For example, the Decoupled Software Pipelining (DSWP) approach extracts pipelines from loops in sequential programs [23] with an automatic compiler [19]. Generally, these techniques are closely related to Smith's Decoupled Access/Execute Architecture [26] and earlier to Hoare's Communicating Sequential Processes [10]. However, these techniques rely on special languages and/or hardware to avoid inefficiencies and achieve their performance potential.

In this work, we present FastForward, a *software only* buffering communication mechanism for multi-threaded pipeline-parallel applications running on multicore hardware with general purpose languages. With FastForward it is possible to construct pipeline-parallel applications with minimal communication overhead, delivering performance improvements proportional to pipeline depth for applications well suited to pipeline parallelism (e.g., network frame processing, multimedia decoding, and certain pointer chasing loops [19]). Note that FastForward may also be applied to other data streaming organizations.

FastForward achieves fast queue/dequeue operations by ensuring that every operation on shared memory is decoupled at the hardware's cache coherence layer and not just the software's algorithmic level. Decoupling makes it possible to hide almost all queue-communication-driven coherence traffic by leveraging the hardware prefetch unit. Additionally, unlike most CLF queues, FastForward guarantees correct operation under strong to the very weakly ordered consistency models used on some general purpose processors (e.g., IBM's Power and Intel's Itanium) and for some special-purpose embedded processors. Further, decoupling at the cache coherence level will increase in applicability as special purpose processors add caches to their designs [11, 18].

Experiments on a 2.66 GHz dual-processor dual-core AMD Opteron 2218 based system show that FastForward operations have an overhead of only 28.5–31 ns, up to 5 times faster than the next best solution allowing for very fine grain ($\leq$200 ns) stages in pipeline parallel applications.

The remainder of this paper is organized as follows. Section 2 provides background and a motivating example. Section 3 describes the implementation and tuning of FastForward. Section 4 presents a proof of correctness. Section 5 presents a detailed evaluation on an AMD Opteron system. Section 6 discusses related work not discussed elsewhere in the paper. Section 7 concludes.

2. Background

This section reviews the three basic parallel structures (i.e., task, data, and pipeline) and motivates FastForward with a network frame processing example.

2.1 Parallel Structures

Recent interest in multi-threaded pipeline parallel applications is driven by the realization that many applications of interest have strict ordering requirements in their computation, making them poor candidates for the conceptually simpler task- and data-parallel techniques. Note, these organizations are primitives and may be composed into more complex organizations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
Copyright © 2008 ACM 978-1-59593-960-9/08/0002…$5.00

Task Parallelism is the most basic form of parallelism and consists of running multiple independent tasks in parallel, usually for relatively long durations (e.g., applications and TCP network connections). This form of parallelism is limited by the availability of independent tasks at any given moment.

Data Parallelism is a method for parallelizing a single task by processing independent data elements in parallel. Bracketing routines fan out data elements and then collect processed results. This technique scales well from a few processing cores to an entire cluster (e.g., MapReduce [6]). The flexibility of the technique relies upon stateless processing routines (filters) implying that the data elements must be fully independent.

Pipeline Parallelism is a method for parallelizing a single task by segmenting the task into a series of sequential stages. This method applies when there exists a partial or total order in a computation preventing the use of data or task parallelism. By processing data elements in order, local state may be maintained in each stage. Parallelism is achieved by running each stage simultaneously on subsequent data elements. This form of parallelism is limited only by inter-stage dependences and the duration of the longest stage.

2.2 Example: Network Frame Processing

Network frame processing provides an interesting case study for pipeline parallelism as such systems are both useful (e.g., intrusion detection, firewalls, and routers) and may exhibit high data rates that stress both the hardware (e.g., bus arbitration) and software (e.g., locking methods). Consider Gigabit Ethernet; the standard specifies support for 1,488,095 frames per second (fps). This means that a new frame can arrive every 672 ns, requiring the software to be capable of removing the frame from the data structures shared with the network card, processing the frame, and potentially inserting it into the output network interface's data structures within 672 ns (approximately 1500 cycles on a 2.0 GHz machine).

Using FastForward, we have built pipeline-parallel applications capable of capturing and forwarding at record breaking, for commodity hardware, rates of 1.428 million and 1.33 million fps, the limit of the evaluation network hardware. These results highlight the ability of general purpose commodity hardware to effectively implement multi-stage pipelines similar to those used on hardware network processors (e.g., Intel's IXP series). In these applications, a 3-stage pipeline was used to achieve approximately a 3x increase in available processing time. Below, we see that since each stage must take no more than 672 ns, this increase requires the very low overhead stage-to-stage communication provided only by FastForward on general purpose machines. Data parallelism is impractical for such applications (e.g., firewalls) as there may be many inter-frame data dependencies. Performance results measured on real hardware are presented in Section 5.7.

A basic forwarding application may be decomposed into three stages (see Figure 1), with each being alloted the full frame computation time period and therefore tripling the available frame manipulation time. The output (OP) and input (IP) stages handle transferring each frame to and from the network interfaces. The application (APP) stage performs the actual application related frame processing. By executing the three stages concurrently it is possible to fully overlap every stage in every time step. The frame processing time can be extended to 4x and beyond if the application stage can be further decomposed. If each stage executes in no more than 672ns per frame, the pipeline will be able to sustain the maximum frame rate on gigabit Ethernet.

Communication overhead is the limiting factor for such fine-grain stages. We found that on a 2.0 Ghz AMD Opteron based system, lock-based queues cost at least 200 ns per operation (enqueue or dequeue). Since each stage has a dequeue and an enqueue operation, this consumes about 60% ($2 \times 30\%$) of the available per-frame processing time. To address this, we developed FastForward to provide a communication primitive costing only 36–40 ns (28.5–31 ns on a 2.66 GHz machine) per operation.

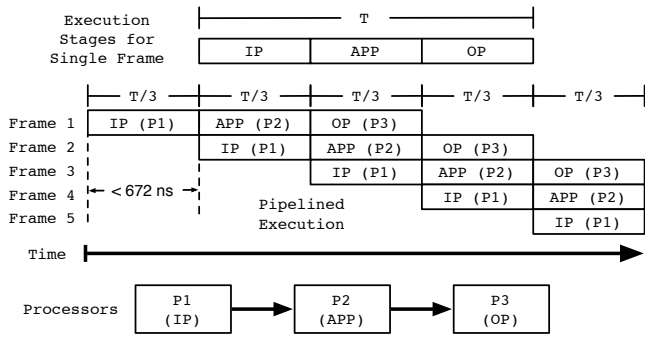


Figure 1. The Frame Shared Memory pipeline.

```
1   enqueue(data) {
2     lock(queue);
3     if (NEXT(head) == tail) {
4       unlock(queue);
5       return EWOULDBLOCK;
6     }
7     buffer[head] = data;
8     head = NEXT(head);
9     unlock(queue);
10    return 0;
11  }

1   dequeue(data) {
2     lock(queue);
3     if (head == tail) {
4       unlock(queue);
5       return EWOULDBLOCK;
6     }
7     data = buffer[tail];
8     tail = NEXT(tail);
9     unlock(queue);
10    return 0;
11  }
```

Figure 2. Locking queue implementation.

3. FastForward

This section presents the design of FastForward and the optimization techniques used to achieve a 3.7–5x improvement over the next best solution. Section 3.1 and 3.2 begin by explaining the shortcomings with existing approaches. Section 3.3 describes FastForward's new cache-optimized single-producer/single-consumer concurrent-lock-free (CLF) queue to overcome the bottlenecks in prior approaches on cache-coherent systems. Section 3.4 describes how to further optimize FastForward's performance by temporally slipping the producer and consumer. Section 3.5 describes how a hardware prefetch unit may further improve performance. Section 3.6 discusses large payload support. We begin with a baseline discussion of traditional lock-based queues.

3.1 Lock Based Queues

Efficient pipeline parallelism requires that the buffering communication mechanism used to provide core-to-core communication provide the smallest overhead possible. In the network frame pro-

```
1  enqueue_nonblock(data) {
2    if (NEXT(head) == tail) {
3      return EWOULDBLOCK;
4    }
5    buffer[head] = data;
6    head = NEXT(head);
7    return 0;
8  }

1  dequeue_nonblock(data) {
2
3    if (head == tail) {
4      return EWOULDBLOCK;
5    }
6    data = buffer[tail];
7    tail = NEXT(tail);
8    return 0;
9  }
```

Figure 3. Lamport's queue implementation.

```
1  enqueue_nonblock(data) {
2    if (NULL != buffer[head]) {
3      return EWOULDBLOCK;
4    }
5    buffer[head] = data;
6    head = NEXT(head);
7    return 0;
8  }

1  dequeue_nonblock(data) {
2    data = buffer[tail];
3    if (NULL == data) {
4      return EWOULDBLOCK;
5    }
6    buffer[tail] = NULL;
7    tail = NEXT(tail);
8    return 0;
9  }
```

Figure 4. FastForward queue implementation.

cessing example (Section 2.2), the communication overhead for two operations must be significantly less than 672 ns. Traditional locking queues are inappropriate for such fine grain stages as the overhead is substantial. Again, for the network processing example, the overhead is greater than 60% of the available per-stage processing time on a 2.0 GHz AMD Opteron and 52% of available time on a 2.66 GHz AMD Opteron.

Locking queues have two basic sources of overhead. The first is algorithmic; notice that the lock operations in Figure 2 strongly couple the producer and consumer. Even when the consumer is reading an earlier enqueued element, the producer cannot enqueue an element into a different buffer slot. The second overhead comes from interaction with cache-coherent multiprocessor architectures. In particular, both the enqueue and dequeue routines read the constantly updated buffer and head/tail slots causing the cache lines containing the data to alternate between the modified (M) and the shared (S) states, with each S to M transition resulting in a new coherence transaction [21]. This implicit synchronization is expensive on small multiprocessors and even more so on large ones.

3.2 Lamport's CLF Queue

Lamport proved that, under sequential consistency, the locks could be removed in the single-producer/single-consumer case (Figure 3), resulting in a concurrent lock-free (CLF) queue. Thus, Lamport's CLF queue requires no explicit synchronization between the producer and consumer [7, 13] decoupling the two at the algorithmic level. Furthermore, any implicit synchronization at the memory layer in the lock implementation is also eliminated. In comparison to the code in Figure 2, this results in a 33% speedup on the 2.66 GHz AMD Opteron mentioned earlier.

However, there still exists an implicit synchronization between the producer and consumer at the memory layer as the control data (i.e., head and tail) is still shared. Thus, on modern cache coherent systems, Lamport's CLF queue still results in cache line thrashing across caches. Furthermore, while Lamport's algorithm was proven correct under sequential consistency [7, 13], it fails under many weaker consistency models. Supporting weaker consistency models may require the addition of potentially expensive fence operations to ensure correct ordering between the data writes (i.e., buffer) and the control writes (i.e., head/tail).

While other CLF data structures have been extensively studied since Lamport's algorithm [12, 13, 15–17, 22, 28, 29], these works have focused on improving the performance of the more general but more difficult multiple-producer/multiple-consumer variants. Handling the ABA problem [16] inherent in these variants requires a linearizable [9] multi-operation mechanism to correctly ensure all parties agree on the order of transactions. Section 6 discusses these approaches in more detail and explains why they are slower than Lamport's algorithm in the single-producer/single-consumer case.

Since pipelines can be implemented with single-producer/single-consumer queues, Lamport's algorithm is the best known CLF queue for pipeline-parallelism and is used as the baseline for the evaluation of FastForward. Note further that many data-flow graphs can also be implemented with single-producer/single-consumer queues, widening applicability of techniques like FastForward.

3.3 The FastForward CLF Queue

FastForward's contribution is an algorithm that provides improved performance over Lamport's queue while operating correctly on a wider range of memory consistency models (strong to very weak). Performance is improved by tightly coupling control and data into a single operation, making it possible for the producer and consumer to operate independently when there is at least one data element in the queue, unlike Lamport's queue. In practice at least a cache-line's worth of entries is needed for best performance. Coupling control and data into a single operation makes the algorithm linearizable [9], a key element in our proof of correctness (Section 4).

Figure 4 shows pseudo-code for the non-blocking enqueue and dequeue operations. Careful reading shows that for each operation only two memory locations are accessed, the index into the buffer and the appropriate buffer entry. This differs from Lamport's queue as the buffer entry itself is used to implicitly indicate full and empty queue conditions. Control is reified by defining a known value indicating an *empty* slot, `NULL` suffices for pointers.

This optimization is critical as it permits stages to keep the cachelines containing the indices in the modified (M) state [21] as they are not shared and therefore thread local. This means that the head or tail indices may always be cache resident and never incur a penalty to make them coherent.

3.4 Temporal Slipping

Avoiding cacheline thrashing is critical for FastForward, and CLF data structures in general. This section describes how FastForward temporally slips the producer and consumer to avoid thrashing by ensuring that enqueue and dequeue operations operate on different cachelines. In practice, it is sufficient to ensure with standard queuing techniques that a cacheline's worth of entries are always available for the consumer instead of the standard approach of ensuring only a single element is available to avoid under-runs. This section considers slip with pipeline stages that are balanced (i.e., have the

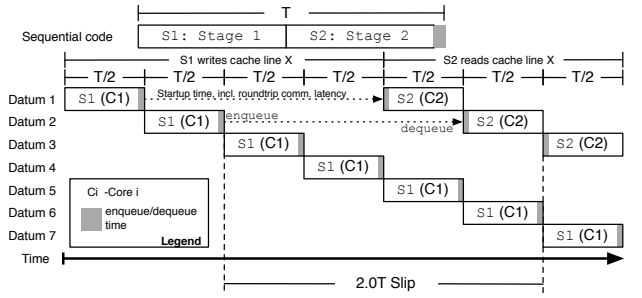


Figure 5. Timing of a slipped pipeline.

```
1   adjust_slip() {
2     dist = distance(producer, consumer);
3     if (dist < DANGER) {
4       dist_old = 0;
5       do {
6         dist_old = dist;
7         spin_wait(avg_stage_time * ( (GOOD+1) - dist) );
8         dist = distance(producer, consumer);
9       } while (dist < GOOD && dist_old < dist);
10    }
11  }
```

Figure 6. Example slip adjustment routine.

same duration per datum) on average; Section 3.4.1 describes an approach to handling unbalanced stages.

Consider a pipeline where stages are working in lock step with no buffered entries separating them. In this scenario the buffer accesses will cause the cacheline containing the buffer entry to bounce between the caches incurring penalties for the coherence traffic just as in Lamport's queue with the head/tail comparisons. Thrashing can be eliminated by recognizing that a pipeline's goal is to maximize throughput, therefore we may reasonably exchange latency for performance.

Further, it is possible to reach the theoretic minimum number of compulsory cacheline transfers with the temporal slipping approach described below. The minimum number of compulsory cacheline transfers, to be explicit, is one per cacheline, so a platform with 64B cachelines transferring 64-bit words should incur only 1 transfer for 8 accesses in the ideal case. The same platform with 32B lines should only need a transfer every 4 accesses.

Figure 5 shows a slipped two stage pipeline on a machine with four entries per cacheline. In this scenario, the consumer stage is delayed by four iterations to permit the producer stage to enqueue four data elements. Once the first cacheline is filled, the producer and consumer run simultaneously with a throughput that is still 2x the non-pipelined version.

Section 5.5 demonstrates the performance improvement of temporal slipping. In effect, temporal slipping achieves the performance advantages of batch transfers (e.g., cache locality and reduced contention) without additional code or data structures.

3.4.1 Slip Maintenance

Initializing and maintaining temporal slip is relatively straightforward provided that any long tail of the stage duration distribution can be measured and bounded. We believe this to be a fair assumption as any high performance streaming application must handle this case as a long tail will result in unacceptably high latencies. Short stages can be readily padded to match longer stages by spinning on the equivalent of the x86 TimeStampCounter. The steps are as follows:

1. Every queue is oversized to ensure it is impossible to thrash a queue by filling it. In our evaluation we ensure this by restricting the number of available external data buffers.
2. All stages upon receiving their first buffer spin until an appropriate amount of slip/queuing is established (see Figure 6). In the example code, `DANGER` represents the point at which slip is likely to be lost, for our system we chose two cachelines of separation or 16 entries. Similarly, an appropriate amount of slip is represented by `GOOD`, and was set to 6 cachelines or 48 entries. These values may need tuning depending on a particular applications variance in stage duration, as per standard queuing practice.
3. If the application's per-stage work time exhibits a non-zero mean variation in nominal work time, or has unbalanced stages, each stage can maintain slip by periodically using the same algorithm used to initially establish slip. The frequency of adjustment needs to be proportional only to the portion of the distribution (i.e., tail) that is not self-stabilizing. In our experiments we ran the slip algorithm once every 64 iterations, or when a dequeue failed, which was sufficient to control the variation while having $\approx$ 1ns of amortized overhead. Notice that the algorithm has an escape path to ensure forward progress is made when there are gaps in the data stream.

This algorithm embodies a primitive yet effective form of distributed control. The key is to ensure the queues start in the stable fast state with every stage slipped. Then it is only necessary to ensure that each queue remains in the stable fast state. This is accomplished by ensuring that each stage on the average does not outrun its producer (step #3 above).

3.5 Prefetching

Temporal slipping provides an opportunity for a hardware prefetch unit to minimize the impact of compulsory cacheline miss penalties by transparently prefetching cachelines into the L2 data cache. Experiments show that, with the AMD Opteron 2218's stride prefetcher, only 1.5% of cacheline accesses were not serviced by the L1 or L2 data caches. Results are detailed in Section 5.5.

Prefetching provides additional potential for performance improvement when passing references to external buffers; discussed in the following section. Content prefetching can be used to hide the transfer cost of transferring such indirect blocks by finding pointers in cachelines and transparently moving them to the local L2 cache [4]. This will cause no additional cacheline thrashing as the data block must be complete before its reference is inserted into the queue. Alternatively, a cache-to-cache push mechanism could be implemented to preemptively transfer the data as done in Intel's work on ETA [24].

3.6 Large Payloads

FastForward as described assumes that every data element can be transfered in a single linearizable write. Larger transfers are possible by dividing each datum into a series of linearizable writes, or by transferring a reference to an external buffer via the queue. Transferring a reference is ideal as communication overhead is incurred for each transfer. On processors with program ordered remote stores (e.g., x86), FastForward operates correctly as writes from remote processors are visible in the writer's program order.

On processors with weaker consistency models the algorithm may need to be slightly modified to introduce a memory store-fence before the reference is written into the queue's buffer (enqueue_nonblock, line 5, Figure 4). Without a store-fence the queue's buffer write could be visible to the remote stage before the payload writes propagate, potentially resulting in a read of stale data. While store-fences may have significant overheads on some

processors (e.g., Power4), this is an unavoidable cost and must be paid with any communication mechanism, whether based on lock-free or locking data structures. Therefore FastForward will still be better for large payloads on all architectures.

4. Proof of Correctness

The intuitive notion of correctness for the point-to-point FastForward queues is that the consumer dequeues values from the queue in the same order the producer enqueued them. However, stating this fact formally is more complicated because modern processors do not execute code in-order, nor are remote writes necessarily seen in the program order of the writer. With this in mind, a more precise statement to prove the correctness of FastForward is to show that "in the program order of the consumer, the consumer dequeues values in the same order that they were enqueued in the producer's program order."

To formalize this notion, we will *not* reason about the order in which operations actually execute; instead we will reason about possible operation orderings whose results would be indistinguishable from the values read and written during execution. Using this reasoning, this section proves that FastForward is correct, to the best of our knowledge, on all general-purpose shared-memory multiprocessor hardware.

The proof will proceed as follows.

1. Assumptions will be defined. In particular, the proof assumes a cache coherent system in which aligned stores are linearizable [9][1], potentially out-of-order processors preserve the illusion of program order for single threads, and speculative stores are not visible to remote loads until they are non-speculative[2]. To the best of our knowledge, every modern general-purpose processor satisfies the above criteria.
2. The proof then shows that in every execution, based on the definition of cache coherence, values read and written to *a particular buffer location* (e.g., `buffer[i]`) are indistinguishable from the values that would have been read or written if all read and write operations to this location executed in the program order of the readers and writers of the location.
3. From the above, the proof shows that for a given `i`, in the consumer's program order, the consumer reads elements from `buffer[i]` in the order in which they were written by the producer in the producer's program order.
4. From this the proof then shows that because of the guarantees of a correct potentially out-of-order processor for single-threaded applications, results of execution are indistinguishable from an execution in which both the producer and consumer iterate over the buffer slots in queue order in their respective program order.
5. Finally the proof shows that the prior two statements mean that the values produced in an actual execution are indistinguishable from an execution in which "in the program order of the consumer, the consumer dequeues values in the same order that they were enqueued in the producer's program order."

Note that any features that change memory orderings seen at runtime but do not violate the assumptions outlined above (e.g., prefetching) are no cause for concern; the proof still holds on such systems since it will only depend these assumptions.

[1] Note that this is not the same as a linearizable memory subsystem; the requirement is simply that each store is seen in its entirety by a processor, or not at all.

[2] Note that this condition is stronger than the statement that speculative stores not appear on the memory interface. However, the latter is sufficient in the absence of techniques such as value prediction (see Hill et al. [14])

4.1 Single Processor Execution

We begin by defining some basic notation and formalize our intuitive notion of a *correct* processor (i.e., one that preserves the illusion of program order for a single thread). Without loss of generality we assume no instruction reads from a location it writes; if this is the case, one can split the instruction's effects into multiple operations.

DEFN 1 (Access). *An access is a tuple $a = (l, v)$ where the value v is read or written from location l.*

Location l can be either a memory location or a register location (e.g., read of the value 5 from general purpose register 3 is denoted (`r3`,5)). When reasoning about multiprocessor systems, we assume that registers are processor local (i.e., not shared).

An operation corresponds to the notion of a dynamic instruction instance. Below, let PC denote the program counter value associated with an operation's corresponding static instruction. Let seq be a number that uniquely identifies operations from the same PC. Let writes be the set of write accesses performed by an operation and let reads be the set of read accesses. cpuid is a an identifier that identifies which CPU, and thus which set of registers, the operation is accessing when reasoning about parallel programs. We introduce cpuid here to allow the definitions to carry through our correctness proof for FastForward.

DEFN 2 (Operation – Dynamic Instruction). *An operation is a tuple $o = (\mathrm{seq}, \mathrm{cpuid}, \mathrm{PC}, \mathrm{writes}, \mathrm{reads})$.*

The following formalizes the notion of an execution of a program, which results in a set of operations.

DEFN 3 (Execution). *An execution, E, is a set of operations.*

An execution is a set and not a sequence because reasoning will be on orders whose results are indistinguishable from those of the actual execution, not on the execution order.

The following definition captures the notion that the results of a potential order of execution are indistinguishable from the results of actual execution.

DEFN 4 (Execution consistent with an Order). *An execution, E, is consistent with an ordering of its operations[3], σ, if and only if for every operation $o \in E$, if o reads a value v from location l (i.e., $(l, v) \in \mathrm{reads}$ for o), v is the value written by the latest writer to l in σ, or l is not initialized before o in σ.*

More formally, E is consistent with σ if and only if for every operation $o \in E$ with a read access (l, v) (1) there exists a unique largest (according to σ) operation, o_w, such that $o_w <_\sigma o$ and o_w has a write access (l, v), or (2) there does not exist any $o_w <_\sigma o$ where o_w has a write access to location l.

Intuitively, a correct uniprocessor executes instructions so that the results of execution are indistinguishable from execution in program order. We now have enough formalism to define this notion.

DEFN 5 (Correct Processor). *A processor is correct for any process P if for every execution, E, of P there exists a total order σ that is consistent with E and has all operations in the program order of P.*

In this definition, the existence of σ for any execution is what implies that the values produced by the execution of the CPU are indistinguishable from those that would be produced had the program run in program order.

[3] An ordering, σ, is an anti-symmetric, reflexive, and transitive relation on some set Ω. If the order σ states that $a \in \Omega$ comes before $b \in \Omega$ we write $a \leq_\sigma b$. If a must not equal b we write $a <_\sigma b$.

To extend this definition to multiple processors, we simply restrict the program order requirement to order operations only from *one* cpu, C (i.e., program order is only enforced for $\mathrm{cpuid} = C$ for every o). With this extension to Definition 5, one can reason about programs on a given CPU as if they executed in program order and generate correct proofs. However, the definition *does not* guarantee that there exists some order σ that is consistent with E in which program order for two or more processors is preserved. In other words, when reasoning about program order on the local CPU using Definition 5, there are no guarantees about the order in which operations from remote CPUs will be observed. Imposing such orders is the domain of the multiprocessor memory model.

4.2 Correctness of FastForward

For the proof we assume a point-to-point connection between a producer and consumer. The model is that of a single producer repeatedly calling `enqueue_nonblock` and a single consumer repeatedly calling `dequeue_nonblock`. The producer only considers the value sent if `enqueue_nonblock` returns 0. The consumer considers a value read only if `enqueue_nonblock` returns 0. A return of `EWOULDBLOCK` means the producer (or consumer) will retry the call.

As mentioned earlier, the proof will rely only on the fact that (1) each potentially out-of-order CPU is correct, (2) stores are not visible to remote processor loads until the stores are no longer speculative, (3) the multiprocessor has a coherent memory, and (4) aligned word-sized stores are linearizable [9]. We will call these assumptions M.

The following theorem formalizes the correctness criterion for FastForward under the above assumptions.

THEOREM 1. *Under machine assumptions M, for any execution E, there exists a total ordering, σ, of the operations in E such that (1) E is consistent with σ, (2) σ contains the operations from the consumer in program order, (3) such that the values read by* `dequeue_nonblock` *in this ordering are those written by the producer, and (4) in σ, the values dequeued by the consumer are dequeued in the same order they were written in the program order of the producer.*

First we show that in program order for the consumer routine, the consumer's `dequeue_nonblock` routine (see Figure 4), for a given buffer location, will read the values written to that location in the order they are written in the program order of the producer routine.

LEMMA 1. *Under the aforementioned machine assumptions, M, for any buffer location* `buffer[i]` *in the word-aligned word-size buffer* `buffer`*, the enqueue code in Figure 4 will only write a new value to* `buffer[i]` *after the entry has been read by the dequeue routine in Figure 4. Furthermore, the dequeue routine will always read the value written by the enqueue routine before clearing the contents of* `buffer[i]`.

Proof To reason simultaneously about the program order of writes to `buffer[i]` in both the enqueue and dequeue routines, we invoke the definition of cache coherence cited as weakest and most common by Adve and Gharachorloo [1]. They define a system to be cache coherent if every execution of the system is cache coherent.

DEFN 6 (Cache Coherent Execution). *An execution, E, is cache coherent if and only if, for each memory location l there exists a total order σ_l that orders the set of memory operations that access location l, call this set E_l; such that (1) σ_l is consistent with E_l, (2) for each processor, σ_l has each memory operation on l in program order for the process on that processor (write serialization), and (3) all processors eventually see all writes to l (write propagation).*

Unlike Definition 5, the above requires the reads and writes to l from each processor appear in program order. Note that the order is only over the memory operations that access l because coherence is only part of the memory model.

By Lemma 1, there exists a σ_l that has the memory operations to any given buffer location `buffer[i]`, from both the producer and consumer, in program order. Thus we can reason simultaneously about program order for both the `enqueue_nonblock` and the `dequeue_nonblock` on this location.

Notice that in σ_l `enqueue_nonblock` only writes to the location `buffer[i]` when it contains a `NULL`. Furthermore, only `dequeue_nonblock` will write `NULL` to `buffer[i]` and, since speculative remote stores not visible, it will only do so only after it sees that `buffer[i]` is not NULL. Finally, `dequeue_nonblock` will only write NULL after reading the contents of `buffer[i]`. This completes the proof of Lemma 1.

From the above, we have the following, now obvious, corollary

COROLLARY 1. *In the program order of the consumer, the routine* `dequeue_nonblock` *reads values out of* `buffer[i]` *in the order in which they are written by* `enqueue_nonblock` *in the program order of the producer.*

We can now prove Theorem 1. Note that the only shared variable between the sender and receiver is `buffer`. Using definition 5, we know there exists an order σ_c that is consistent with any execution, E, and has all the consumer's operations in program order. In σ_c, the slots of `buffer` are read, and cleared in ascending order with wrap around (thus the use of `NEXT(tail)` instead of `tail++`). Now consider the corresponding order σ_p for the producer. Here, the slots of `buffer` are written in the same order. Furthermore, note that the producer and consumer start with `head==tail`.

Now from Corollary 1, we have that each buffer slot `buffer[i]` is read by the consumer, in its program order, in the order in which it was written by the producer in its program order. Furthermore, in the respective program orders the `enqueue_nonblock` and `dequeue_nonblock` routines access the buffer slots in the same order. Thus, in the consumer's program order, values are read in exactly the order they were enqueued in the producers program order. $\mathcal{QED}$

4.3 Leveraging Weak Ordering

Note that the above proof is subtle. The proof of correctness only shows that FastForward has the data transmission properties of a queue. The proof says nothing about the synchronizing properties of lock-based queues. Consider the case where one uses FastForward to enqueue pointers that will be read and dereferenced to access the payload data in the dequeuing stage. Here, there may be a problem on some memory consistency models because the dequeue routine may read the value of the pointer out of the queue and dereference the pointer *before* the dequeuing stage sees the writes that define the payload data. In this case, the dequeue routine will read a correct pointer but get the wrong (i.e., stale) payload data.

By separating the data transmission properties from the synchronization properties FastForward enhances queue performance on machines with weakly ordered memory models. On some memory models (e.g., sequential consistency, total store ordering, and the x86 model) FastForward also has all the necessary synchronization properties of a queue. On other models (e.g., Itanium and Power4, 5, and 6), if synchronization is needed, a store fence before an enqueue is needed.

5. Evaluation

This section presents a performance evaluation of an implementation of FastForward. This performance evaluation shows that FastForward's performance is $\approx$ 3.7x better than Lamport's queue on

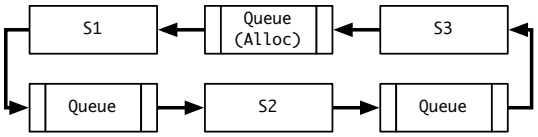


Figure 7. Looped pipeline configuration.

real hardware, and that FastForward works for real applications. Results show that FastForward's performance is work load invariant, queue size invariant, insensitive to core placement, and is robust to variations in stage processing lengths.

Section 5.2 evaluates the performance across queue size and workload. Section 5.3 evaluates the effect of fences. Section 5.4 shows that FastForward can be robust with respect to variations in stage duration and unbalanced stages. Section 5.5 shows the cache benefits of temporal slipping and prefetching. Section 5.6 finds that there is no penalty for communicating off-die as compared to on-die. Section 5.7 concludes by reporting an experiment in which FastForward was applied to a network frame processing application.

5.1 Evaluation Methodology

The methodology used to evaluate FastForward in this paper consists of performance feature isolating benchmarks and an example application scenario in which FastForward was applied to network frame forwarding. The performance isolating benchmarks compare against Lamport's queue as it is the gold standard for the special case of single-producer/single-consumer queues (Section 3.2).

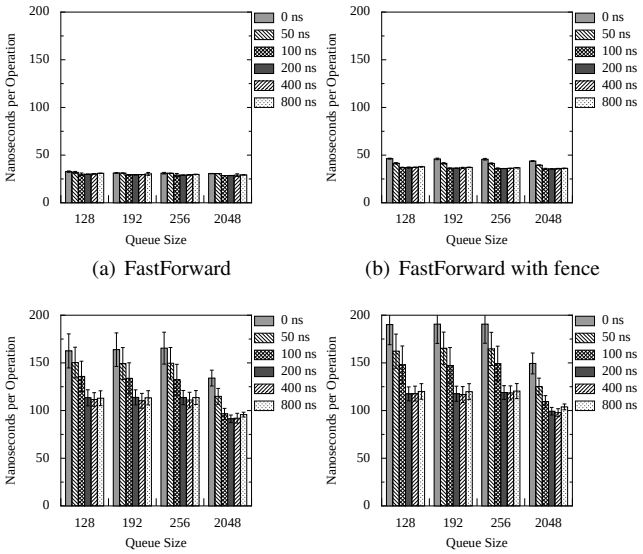
Each plotted data point, unless specified otherwise, consists of the sample mean and sample standard deviation of 100 trials with 1,000,000 iterations.

Each trial consists of two or three stages[4] forwarding in a looped pipeline configuration (Figure 7). Each stage iteration consists of dequeuing a value from the input queue, spinning on the cycle accurate x86 TimeStampCounter for a specified number of cycles, and then enqueuing the value on the output queue. Spinning is used to approximate work in a controlled fashion to isolate queue performance. Robustness is verified in Section 5.4 by adding noise to the stage duration for each iteration and employing the slip maintenance algorithm. To ensure that stages start slipped, each stage spins until slip is established after receiving the first datum. To ensure the queues don't fill up behind a slow stage, every queue is initialized with no contents, except for the first (Alloc) which is populated with $QueueSize - 16$ elements (Section 3.4.1). This configuration demonstrates message passing in a looped pipeline with pointer-based external payload buffers.

For each of the experiments both algorithms were evaluated with four different queue sizes (128, 192, 256, and 2048) and six different work period durations (0 ns, 50 ns, 100 ns, 200 ns, 400 ns, and 800 ns). These parameters measure memory footprint and any temporal effects caused by stage duration (work load).

The test hardware for the isolated performance analysis consisted of an AMD Opteron system with an ASUS KFN32-D SLI R motherboard equipped with two dual-core 2.66GHz AMD Opteron 2218s and 2GB (4x512MB DDR2 667) of memory in a two-way ccNUMA configuration. The referenced networking example used a similar environment with 2.0 GHz AMD Opteron 270s. For completeness we note that while threads were pinned to processors and no operating system services were used during the trails, FreeBSD 6.2 was used to evaluate the queues and FreeBSD 5.5 for the network forwarding example.

[4] Our evaluation hardware only had 4 processors, and one must be reserved for OS activity for accurate timing measurements.



(a) FastForward (b) FastForward with fence

(c) Lamport (d) Lamport with fence

Figure 8. Queue comparisons (baseline and with fences) [stage duration vs. queue size].

5.2 Performance

Figure 8 compares the performance of FastForward to Lamport's queue with respect to work load and queue size with a three-stage pipeline. For temporal slip, the FastForward queue is started in the fast stable state (i.e., things start slipped). Section 5.4 is the only section that presents results with the slip maintenance algorithm from Section 3.4.1. Each graph shows the average per operation costs for a single enqueue or dequeue operation.

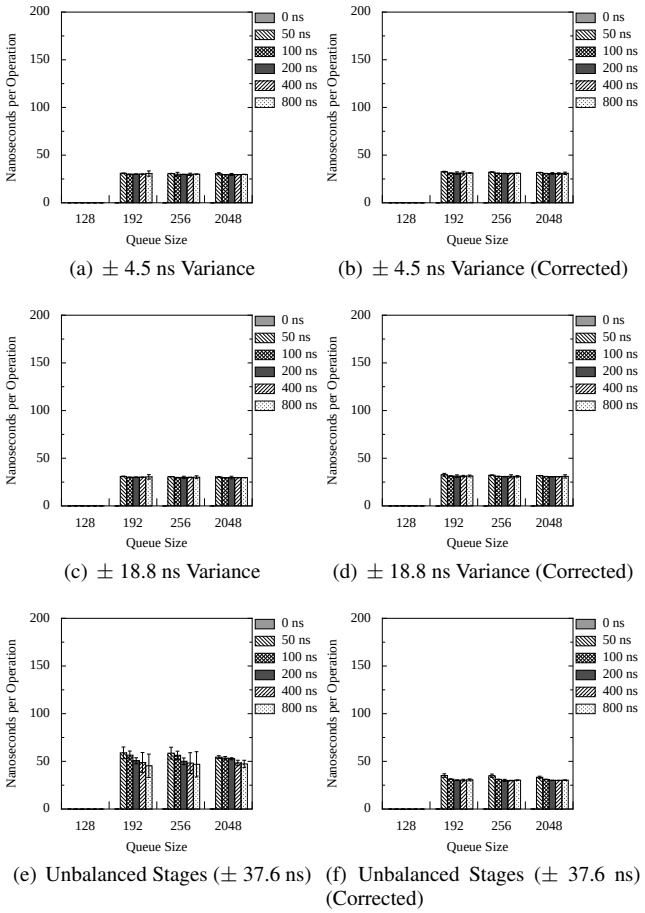
The main observation is that FastForward is insensitive to both queue size and simulated work load while Lamport's queue is not (Figures 8(a) & (c)). FastForward takes on average 28.5–31 ns per queue operation with typical standard deviations less than 1 ns. FastForward is 4.4–5x faster than Lamport's queue for the smallest work loads (the hardest) and 3.7x faster for longer workloads. Further, Lamport's queue exhibits large variations in the average queue cost between executions showing the algorithm to not be stable on the evaluated system.

5.3 Performance with Memory Fences

This section briefly evaluates the potential impact of adding a store-fence to both FastForward and Lamport's queue. Recall from Section 3.6 that fences are necessary on some machines with weak memory consistency models. Figure 8 suggests that while adding a fence may slow down the performance of the queues, it does so in a uniform manner with no impact on algorithmic performance. The impact of fences varies by platform, but for x86 remote writes guaranteed to be seen in program order and thus store fences are inexpensive.

5.4 Robustness

For maximum performance, it is critical that FastForward maintains temporal slip, which becomes difficult when work time may vary. This section evaluates the robustness of the FastForward algorithm and the slip maintenance algorithm described in Section 3.4.1 against variance in work times. The evaluations show that: (a) FastForward is stable when slip is properly initialized, (b) the slip maintenance algorithm only adds a nanosecond of overhead, (c) the slip maintenance algorithm can compensate for unbalanced stages.



(a) ± 4.5 ns Variance (b) ± 4.5 ns Variance (Corrected)

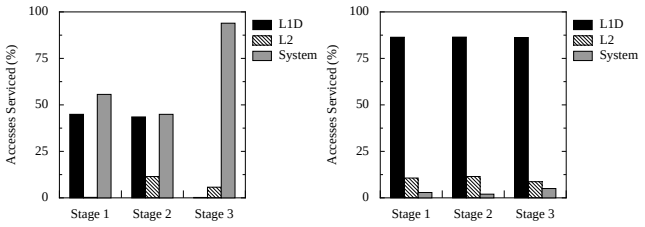
(c) ± 18.8 ns Variance (d) ± 18.8 ns Variance (Corrected)

(e) Unbalanced Stages (± 37.6 ns) (f) Unbalanced Stages (± 37.6 ns) (Corrected)

Figure 9. Evaluation of robustness with and without automatic Slip Maintenance [stage duration vs. queue size]. Note: due to insufficient buffers, queues with < 144 entries were not evaluated.

We evaluate the robustness of the FastForward algorithm by randomly varying the duration of work for each datum processed. First, we use two zero-mean adjustments to a nominal work time, ± 4.5 ns and ± 18.8 ns. Figure 9(a) shows the behavior of FastForward with a small variance in stage duration, corresponding to a well tuned applications that suffers only a few cache misses or pipeline stalls. For this case 50% of the iterations were run with the labeled work load time and the remaining 50% was split evenly between labeled nominal work load plus 4.5 ns and the nominal minus 4.5 ns. Figure 9(b), labeled "Corrected", shows the same experiment with the slip maintenance code enabled. Figure 9(c) and (d) show a similar but wider distribution reflecting a less well tuned application where 50% of the iterations were run with the labeled work load, 40% were run with ± 9.4 ns variation, and the remaining 20% were run with ± 18.8 ns. As expected, both scenarios were self-stable; no slip-maintenance routine was required to maintain performance. Adding the slip maintenance routine described in Section 3.4.1 added ≈ 1 ns of overhead (Figures 9(b) & (d)). There is also ≈ 1 ns of overhead from the code to introduce the variance.

Most importantly from a robustness perspective, the system was evaluated with unbalanced queues as it is not always possible or desirable to manually balance the queue stages. We repeated the above experiment, this time with a loop of three sequential stages with fixed durations of 0, $+37.6$ ns, and -37.6 ns. Example: S1 = 100 ns + 0, S2 = 100 + 37.6 ns, and S3 = 100 - 37.6 ns. Notice that since S2 is always the slowest stage, without slip maintenance, the data will pile up in S2's input queue, leaving S1's and S2's input queues empty. That means, in this case, almost every access of S3's output queue is a cache miss since S1 grabs the buffer cache line while it waits for S3's output. Figure 9(e) and (f) show that without any slip maintenance (Figure 9(e)) the cost of each queue operation for S2 increases by 15–28 ns compared to the same experiment with slip maintenance turned on (Figure 9(f)). Note that even with this penalty, FastForward is still faster than Lamport's queue. This confirms that, within reason, it is possible to robustly compensate for variance and unbalanced stages.



(a) Not Slipped (b) Slipped (Correcting)

Figure 10. Example cache behavior from Figures 9(e) & (f).

5.5 Cache Behavior

Figure 10 compares the extremes of cache behavior for FastForward based on hardware performance counter measurements of the number of L1 data cache accesses and the number of misses serviced by the L2 cache or by the system (both cache-to-cache transfers and transfers from main memory). The graphs depict the cache measurements for a run from the experiments whose results were depicted in Figures 9(e) & (f). Figure 10(a) shows the behavior with no slip maintenance enabled (and thus no slip) and Figure 10(b) shows the same experiment with the slip maintenance enabled.

As expected the non slipped case (Figure 10(a)) performs poorly with respect to cache accesses. Notice that a significant portion of the data access are serviced directly by the cache-to-cache transfers (labeled as system). Further notice that for the third stage, $\approx 94\%$ of the data access are serviced by the system.

Compare this to the same configuration in which the auto slip maintenance routine is enabled (Figure 10(b)). In this case even though the stages are still unbalanced, the code rebalances the stages so that we see the predicted 8:1 reduction in L1 data cache misses (see Section 3.4). Further notice that the prefetcher successfully achieves another 8:1 reduction in cache misses at the L2 level. This confirms that slip is critical for for the best performance in FastForward. Furthermore, these results suggest, as will be shown below, that FastForward will be insensitive to off-die communication for the evaluation scenarios because of this prefetching.

5.6 Performance On and Off Die

This section isolates and evaluates the performance of FastForward for two stages communicating on and off die. Figures 11(a) & (b) show that the performance of FastForward is insensitive with respect to core placement, unsurprising given the cache behavior shown above (Section 5.5).

The data further shows that Lamport's queue is not insensitive and incurs a 10–20 ns overhead when communicating off-die (Figures 11(c) & (d)). This is not surprising as Lamport's algorithm is not decoupled and therefore cannot benefit from prefetching.

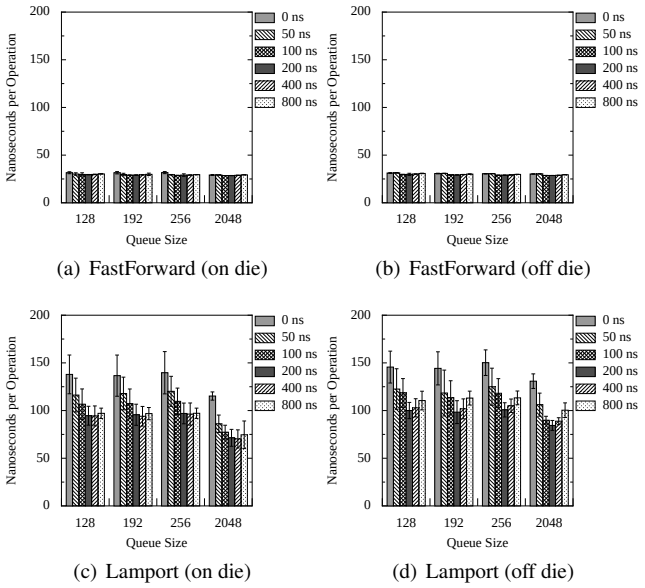


Figure 11. Queue comparison (on-die vs. off-die) [stage duration vs. queue size].

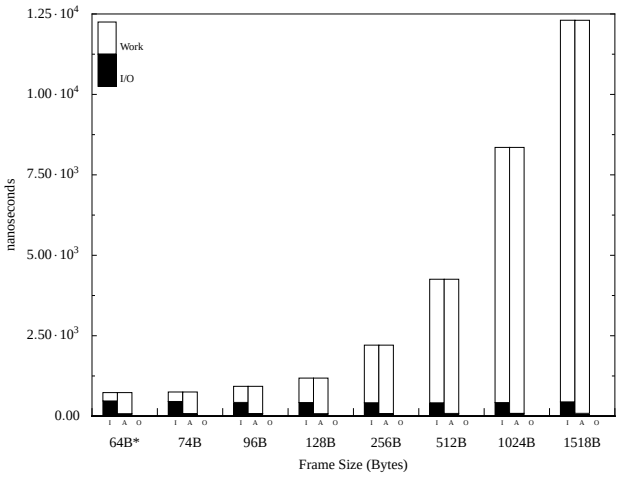


Figure 12. Network frame forwarding.

5.7 Network Frame Processing

We conclude our evaluation by confirming that the performance isolating benchmarks described above are realistic for implementation in applications by reporting an experiment, on gigabit Ethernet, from the soft network processing example described in Section 2.2. The evaluation shows a frame forwarding application in which frames are captured by one processor that may perform additional work on the frame before forwarding (I for input) to a second processor for additional processing (A for application) and finally being forwarded to a third processor which transmits the frame (O for output). Note that no time is measured for the output stage as all application work should be completed before delivered to the output device. Figure 12 shows that the stage-to-stage communication time is small and constant regardless of frame size as expected. Note that the input time for the input stage may appear longer than expected because it includes the time to pull data out of the data structures shared with the network interface. The graph shows that the system is capable of forwarding at 1.36 million frames per second, sufficient for line-rate for all frame sizes $\geq$74bytes. Further analysis shows that the framework itself is capable of higher frame rates. The applicability of the technique is therefore confirmed. This performance is not possible with Lamport's queue as the additional communication overhead exceeds the available time.

6. Related Work

Concurrent lock-free queues have been widely studied in literature [12,13,15–17,22,28,29]. However, with the exception of Lamport's queue [13], the focus of this prior work has been on multiple-producer and/or multiple-consumer (MP/MC) variants. These general purpose queues to date have been limited in performance due to overheads associated with avoiding the ABA problem [16]. Avoiding the ABA problem requires some linearizable [9] mechanism to correctly ensure all parties agree on the order of multi-operation of transactions. Modern processors provide the necessary linearizable operations as hardware primitives, e.g., compare-and-set (CAS) and load-linked/store-conditional (LL/SC). Unfortunately, these primitives implicitly introduce synchronization at the hardware level that are often an order of magnitude slower, even for uncontested cache aligned and resident words, than primitive linearizable stores. Additionally, general purpose MP/MC variants often use linked lists which require indirection, exhibit poor cache-locality, and require additional synchronization under weak-consistency models. Recent MP/MC queues described by Tsigas and Zhang [28], Ladan-Mozes and Shavit [12], Moir et. al. [17], and Scherer III et. al. [25] all improve on previous performance results but retain the need for costly hardware synchronization primitives. Further, note that while the queue described by Tsigas and Zhang [28] bears some similarity to the FastForward design as it uses both an array and *two* NULL values, the algorithms design is focused on an efficient MP/MC queue.

For the special case of single-producer/single-consumer (SPSC) queues, the extra operations or linked data structures in MP/SC solutions are not necessary and hinder performance compared to SPSC-specific solutions such as Lamport's queue and FastForward. Thus, FastForward is designed to optimize, and only support, the important single-producer/single-consumer queue type for use in constructing efficient pipelines. Due to this limited scope, the ABA problem is non-existent since there is only one reader and one writer. Therefore it is possible to avoid the use of synchronizing hardware primitives (e.g., CAS and LL/SC). FastForward further improves upon this by permitting completely decoupled operation by restructuring the queue control code and introducing temporal slip. Thus FastForward avoids per-operation coherence traffic (see Section 3). Lamport's queue [13], on cache-coherent systems, has implicitly coupled control and data variables and therefore cannot provide for fully decoupled operation on modern hardware. Note also that dual-lock queues such as the one described by Michael and Scott can be used in SPSC mode without locks [16]. However, that algorithm uses linked lists suffering the aforementioned performance and correctness issues.

Finally, interesting comparisons can also be made to shared memory synchronization primitives such as the one described by Anderson [3]. Anderson's primitive bears similarity to the design of FastForward as they both optimize performance by minimizing false sharing. However, Anderson's algorithm blocks until success, supports only a single data slot, and fences may be required for basic operation on machines with weak consistency models.

7. Conclusion

This paper presented FastForward, a high-rate core-to-core communication algorithm for instantiating efficient fine grain pipeline-

parallel applications. FastForward is a new cache-optimized single-producer/single-consumer concurrent lock-free (CLF) queue providing fully decoupled operation. By leveraging temporal slipping and hardware cache prefetching, FastForward can enqueue or dequeue pointer sized data elements in 28.5-31 ns, on the tested hardware, 3.7x faster than the next fastest CLF queue implementation and up to 5x faster for very fine grain stages ($\leq$200 ns work duration). The performance was found to be insensitive with respect to work load, queue size, core allocation (i.e., on or off die), and stage work duration variance. A proof of correctness is provided for machines with strong to very weak consistency models. Finally, the efficiency of FastForward is demonstrated for real applications by using it to implement a network processing engine that delivers unprecedented performance on general purpose hardware.

8. Acknowledgments

The authors gratefully acknowledge Joseph Dunn, Todd Mytkowicz, Christoph Reichenbach, Jeremy Siek, and the anonymous reviewers for their comments on previous drafts of this work.

This material is based in part upon work supported by the generosity of AMD Corporation, the University of Colorado Engineering Excellence Fund, and the National Science Foundation under grant CNS 0454404, "Wireless Internet Building Blocks for Research, Policy, and Education."

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] S. Amarasinghe. Multicores from the compiler's perspective: A blessing or a curse? In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 137–137, Washington, DC, USA, March 2005. IEEE Computer Society.

[3] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[4] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–290, New York, NY, USA, 2002. ACM Press.

[5] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–248, New York, NY, USA, 2005. ACM.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04: Proceedings of the Sixth USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, December 2004.

[7] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *SPAA '91: Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–326, New York, NY, USA, 1991. ACM Press.

[8] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, 1992.

[9] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[10] C. A. R. Hoare. Communicating sequential processes. `http://usingcsp.com/cspbook.pdf`, 2004. Electronic edition edited by Jim Davies.

[11] S. Kumar, J. Maschmeyer, and P. Crowley. Exploiting locality to ameliorate packet queue contention and serialization. In *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pages 279–290, New York, NY, USA, 2006. ACM Press.

[12] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. In *DISC '04: Proceedings of the 18th International Conference on Distributed Computing*, volume 3274, pages 117–131. Springer, 2004.

[13] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[14] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 328–337, Washington, DC, USA, 2001. IEEE Computer Society.

[15] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *SOSP '89: Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 191–201, New York, NY, USA, 1989. ACM Press.

[16] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[17] 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 Press.

[18] J. Mudigonda, H. M. Vin, and R. Yavatkar. Overcoming the memory wall in packet processing: hammers or ladders? In *ANCS '05: Proceedings of the 2005 Symposium on Architecture for Networking and Communications Systems*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[19] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO '05: 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[20] G. Ottoni, R. Rangan, A. Stoler, M. Bridges, and D. August. From sequential programs to concurrent threads. *IEEE Computer Architecture Letters*, 5:6–9, 2006.

[21] 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 Press.

[22] S. Prakash, Y. H. Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.

[23] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[24] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, and A. Foong. ETA: Experience with an Intel Xeon processor as a packet processing engine. *IEEE Micro*, 24(1):24–31, 2004.

[25] 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 Press.

[26] J. E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, 1984.

[27] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.

[28] 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 Press.

[29] J. D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 1996.

View publication stats