FastForward for Efficient Pipeline Parallelism

John Giacomoni, Tipp Moseley, and Manish Vachharajani

University of Colorado at Boulder
Technical Report CU-CS-1028-07
April 2007

Dept. of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309-0430

Dept. of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, Colorado 80309-0425

FastForward for Efficient Pipeline Parallelism

John Giacomoni, Tipp Moseley, and Manish Vachharajani

Abstract

High-rate core-to-core communication is critical for efficient pipeline-parallel software architectures. This paper presents the FastForward system, a software-only low-overhead high-rate queue implementation for pipeline parallelism on multicore architectures. FastForward uses an architecturally-tuned domain-specific adaptation of concurrent lock-free queues to provide low-latency and low-overhead core-to-core communication. Enqueue and dequeue times on a 2 GHz Opteron 270 based system are as low as 36 ns, up to 4x faster than the next best solution. FastForward's effectiveness is demonstrated for real applications by applying it to network processing, resulting in record-breaking throughput for commodity hardware. A proof of correctness shows that FastForward works on strong to very weakly ordered consistency models.

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 chip multiprocessors (CMPs) that have multiple cores on a single die. While CMPs are a boon to throughput driven applications such as web servers, single-threaded applications' performance is stagnant. Furthermore, the typical approach to parallelizing software is difficult for general purpose applications [2] as the strategy has been to find, extract, and run nearly independent code regions on separate processors [14].

Recent work shows that many applications can be parallelized by using pipeline parallelism [11, 12, 14, 16]. In this paradigm, computation is divided into 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 [14] with an automatic compiler [11]. However, these techniques rely on special languages or hardware to achieve their performance gains.

In this work, we present FastForward, a *software only* buffering communication mechanism for multi-threaded pipeline-parallel applications running on commodity multicore hardware with general purpose languages. With FastForward it is possible to construct pipeline-parallel applications with minimal communication overhead, delivering performance improvement proportional to pipeline depth for applications well suited to pipeline parallelism (e.g., network frame processing, multimedia decoding, and certain pointer chasing loops [11]).

FastForward provides fast queue/dequeue operations by using shared-memory regions to construct concurrent lock-free (CLF) [10] queues that eliminate all synchronization operations and enable decoupled operation. Unlike most CLF queues, FastForward guarantees correct operation under strong to very weakly ordered consistency models. For performance, FastForward uses a careful organization and a cache-aware timing strategy allowing the prefetcher to eliminate most compulsory cache misses. Furthermore, since all FastForward state is thread local, both OS process migration and cross-domain (e.g., between kernel and user-processes) operation is handled seamlessly. Experiments on a 2.0 GHz dual-processor dual-core AMD Opteron 270 show that FastForward operations have an overhead of only 36-40 ns, up to 4x times faster than the next best solution allowing for very fine grain ($\leq$200 ns) stages.

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 an AMD Opteron system. Section 6 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 matches for existing task- and data-parallel techniques.

Task Parallelism is the most basic form of parallelism and consists of running multiple independent tasks in parallel. 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 [4]. 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 stresses both the hardware (e.g, bus arbitration) and software (e.g., locking methods). Consider gigabit Ethernet, at the maximum frame rate case there are 1,488,095 frames per second. This means that a new frame can arrive every 672 ns, requiring the software to remove the frame from the data structures shared with the network card, process the frame, and if the frame is being forwarded insert it into the output network interface's data structures within 672 ns (approximately 1500 instructions on a 2.0 GHz machine).

Using FastForward, we have built applications capable of capturing and forwarding at a record breaking, for commodity hardware, rate of 1.428 million frames per second, the limit of the evaluation network hardware (see Section 5.6). To achieve this result on general purpose commodity hardware a 3-stage pipeline was used to increase available per frame processing time approximately 3x. Below, we see that this increase requires the very low overhead stage-to-stage communication provided only by FastForward on general purpose machines. Data parallelism is

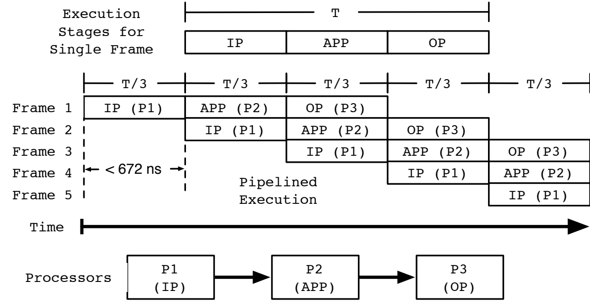


Figure 1. Frame Shared Memory Pipeline

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 5.6. The lessons learned from this example may be generalized to any domain with ordered data.

A basic forwarding application may be decomposed into three stages (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.

Communication overhead is the limiting factor for such fine grain stages. We found that lock based queues cost at least 200 ns per operation (get or put), or ≈60% (2x30%) of the available frame processing time. To address this, we developed FastForward to provide a communication primitive costing only 36-40ns per operation while providing cross-domain communication so the application stages may be in user-space with the I/O stages in the kernel.

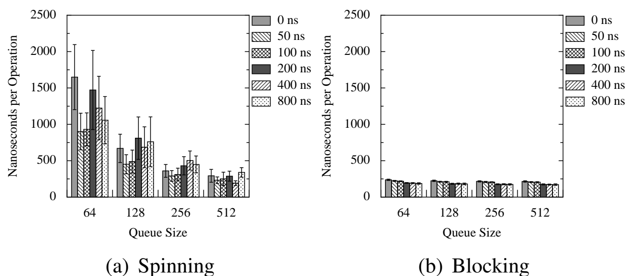
3 FastForward

This section presents FastForward, a software-only high-performance communication primitive that is 2.5x-5x faster than the next best solutions: traditional lock-based queues (Section 3.1) and Lamport's concurrent lock-free (CLF) queue (Section 3.2). Fastforward uses a new optimized single-producer/single-consumer CLF queue. Section 3.3 describes this queue and how it overcomes the bottlenecks in traditional lock-based queues and Lamport's CLF queue. Sections 3.4 and 3.5 describe how to maximize the performance of FastForward on modern memory subsystems with temporal slipping and prefetching. Finally, large payload support is discussed in Section 3.6.

```
1   enqueue_nonblock(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_nonblock(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



(a) Spinning (b) Blocking

Figure 3. Performance of in-kernel locking queues on an AMD system for spinning versus blocking 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 processing 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 proportionately substantial ($\geq 60\%$).

Locking queues (Figure 2) are inappropriate due to two basic limitations. First, the cost of manipulating a locking queue in a non-preemptive environment, such as inside a kernel[1], incurs substantial overhead (Figure 3). Notice that even when a blocking condition variable is used Figure 3(b), the performance is still unacceptable. Second, it is often desirable for stages to be located in different processes, such as the network example with stages in the kernel and user-space. In these situations the only way to block on the condition variable is with expensive system calls (Table 1).

[1]FreeBSD 5.5, our host OS does not support the pinning of user-space threads necessary to eliminate unpredictable OS related stalls

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

Figure 4. Lamport's queue implementation

Architecture	# Threads	Syscalls	ioctl
2.0 GHz AMD Opteron 270	N/A	168 ns	556 ns
2.0 GHz AMD Opteron 270	1	1334 ns	1799 ns
2.0 GHz AMD Opteron 270	2	911 ns	1211 ns
2.0 GHz AMD Opteron 270	3	910 ns	924 ns

Table 1. System-call and ioctl costs. N/A means no thread library is active.

Therefore a lower per operation cost technique is needed to support fine-grain pipeline stages.

3.2 Lamport's CLF Queue

The real problem with lock based queues is the explicit synchronization between the producer and consumer. This synchronization prevents independent operation even if the producer and consumer can independently operate on different entries in the queue. Lamport in 1983 proved that the locks could be removed in the single-producer/single-consumer case (Figure 4), resulting in a concurrent lock-free (CLF) queue requiring no explicit synchronization between the producer and consumer [8]. Notice that there still exists an implicit synchronization between the producer and consumer at the memory layer as the control data (i.e., head and tail) are shared. This dependency, on a system with caching, causes the cachelines containing the head and tail indices to transition between the modified (M) and shared (S) states for every enqueue or dequeue opera-

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

Figure 5. FastForward queue implementation

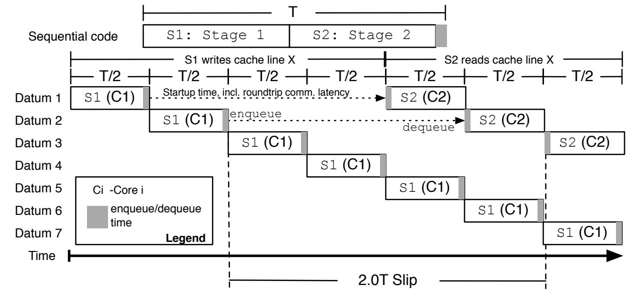


Figure 6. Timing of a slipped pipeline.

tion [13], and therefore thrash the cache coherence protocol.

Lamport's algorithm was proven correct under sequential consistency [5, 8], but not anything weaker, although one can convince oneself that it is also correct under some models (e.g., x86 [7]). Supporting weaker consistency models requires the addition of fences to ensure correct ordering between the data writes (i.e., buffer) and the control writes (i.e., head/tail). While CLF data structures have been extensively studied since Lamport's algorithm, they do not meet our performance requirements as they usually depend on linked lists or double-atomic compare-and-set operations that are not found on most processors [10].

3.3 FastForward's 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. Coupling control and data into a single operation means that the algorithm is linearizable [6], a key element in our proof (Section 4).

Figure 5 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 [13] 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 thrashing of cachelines (i.e., memory hotspots) is critical for FastForward, and CLF data structures in general. Temporally slipping the producer and consumer in FastForward eliminates thrashing by ensuring that enqueue and dequeue operations operate on different cachelines.

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 in FastForward by recognizing that a pipeline's goal is to maximize throughput while minimizing the communication overhead. It is possible to reach the theoretic minimum number of compulsory cacheline transfers by temporally slipping the processing stages to ensure there is never a time when the producer and consumer are operating on the same cacheline. The minimum number of compulsory cacheline transfers, to be explicit, is one per cacheline, so a 64-bit platform with 64B cachelines incurs only 1 transfer for 8 accesses. This reduction in communication overhead incurs only a modest increase in latency. Section 5.4 demonstrates the performance improvement of temporal slipping.

Figure 6 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. Further, by minimizing the communication overhead the time available for work in each stage is increased by minimizing the communication overhead.

Initializing and maintaining this slip is straightforward

given stages of uniform duration and a zero mean distribution of unexpected delays. Balancing the stages so that they are all of uniform duration can be accomplished by spinning on the equivalent of the TimeStampCounter in x86 to match the duration of the longest stage.

3.5 Prefetching

Temporal slipping provides an opportunity for a hardware prefetch unit to minimize the cost of the compulsory per buffer cacheline penalty by transparently prefetching cachelines into the L2 data cache. Experiments show that, with the AMD Operton 270's stride prefetcher, only 1.5% of cacheline accesses were not serviced by the L1 or L2 data caches. Results are detailed in Section 5.4.

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 [3]. This is 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 [15].

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 transfering a reference to an external buffer via the queue. Transferring a reference is ideal as communication overhead must be paid 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 needs to be slightly modified to introduce a memory store-fence before the reference is written into the queue's buffer. Without a store-fence the queue's buffer write could be visible to the remote stage before the payload writes complete, 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, be it based on CLF data structures or mutexes. 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 FastFoward 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 [6], 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.

[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. [9])

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."

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 uninitialized 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' 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.

[3]An ordering, σ, is a 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 $a <_\sigma b$.

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.

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 producer repeatedly calls `enqueue_nonblock` and the consumer repeatedly calls `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 [6]. 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 5), 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 5 will only write a new value to* `buffer[i]` *after the entry has been read by the dequeue routine in Figure 5. 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* `dequeue_nonblock` *routine 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 reciever 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 Corrollary 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 weak 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 an evaluation of FastForward. It shows that FastForward's performance is 2.5x-4x better than Lamport's queue on real hardware, and that FastForward works for real applications. Results show that FastForward's performance is work load invariant, queue size invariant, and insensitive to core placement.

Section 5.2 evaluates the performance across, queue size, workload, and spinning vs. blocking. Section 5.3 evalautes the effect of fences. Section 5.4 shows the cache benefits of temporally slipping and prefetching. Section 5.5 finds that there is no penalty for communicating off die as compared to on die. Section 5.6 concludes by applying FastForward to a full network frame processing application.

5.1 Evaluation Environment

Each parameter is evaluated on looped pipelines with 2 or 3 stages [4] in which the last pipeline stage communicates a message back to the first. We used a looped pipeline to demonstrate message passing with pointer-based external payload buffers. Each stage is identical; all read from their input queue, spin for a specified amount of time (to measure performance with varying workloads), and write the input value to their output queue. The initial stage is identified by filling its input queue with an initial set of payload buffer pointers. Spin time is reliably measured using a cycle accurate TimeStampCounter ($\approx$ 2-4 cycles).

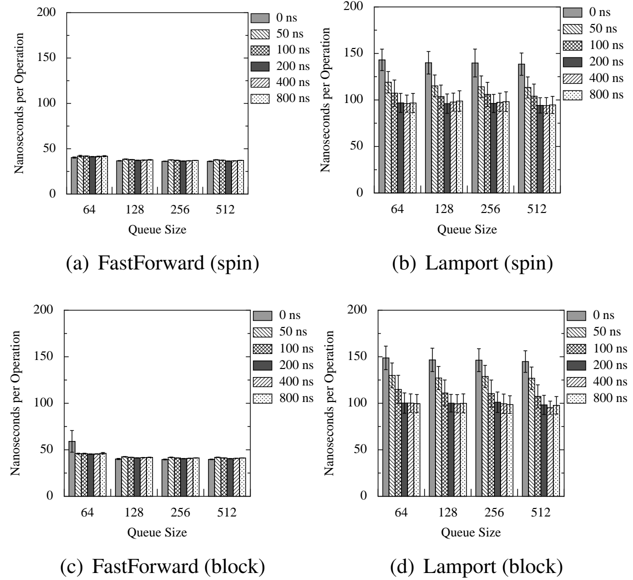
All data points, unless specified otherwise, were generated by taking the mean of 100 executions and calculating one standard deviation. In all executions one million data elements were passed through the pipeline, and the queuing costs were identified by subtracting the time spent spinning from the average per stage time.

The test hardware consisted of an AMD Opteron system with a Tyan Thunder K8SR (S2881) motherboard equipped with two dual-core 2.0GHz AMD Opteron 270s.

5.2 Performance

Figure 7 compares the performance of FastForward to Lamport's queue with respect to work load, queue size, and spinning vs. blocking with a three stage pipeline. Each graph shows the per operation costs for a single enqueue or dequeue with either spinning or blocking behavior; the queue size (number of slots) and work time (different bars) are also varied.

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



(a) FastForward (spin) (b) Lamport (spin)

(c) FastForward (block) (d) Lamport (block)

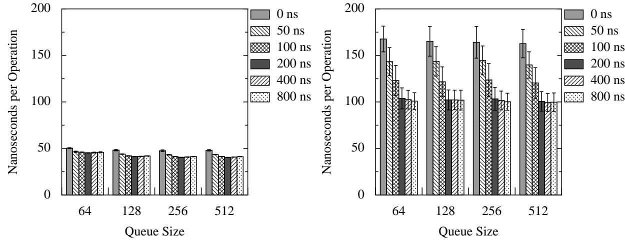
Figure 7. Queue Comparison.

The main observation is that FastForward is insensitive to both queue size and simulated work load while Lamport's queue is not. FastForward, in spinning mode, takes an average of 36-40 ns per operation with standard deviations less than 1 ns for all configurations. FastForward is 3-4x faster than Lamport's queue for the smallest work loads (hardest) and 2.5x faster for longer workloads.

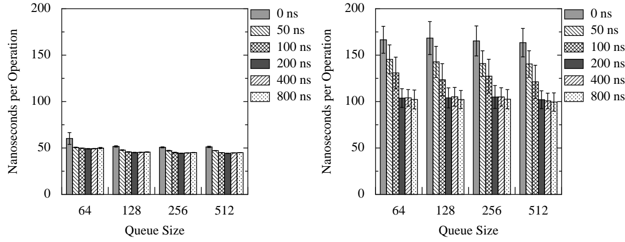
FastForward also demonstrates excellent performance over Lamport's queue while in blocking mode with an average of 39-45 ns with standard deviations less than 1 ns; queue size equal to 64 and work load equal to 0 ns a sole outlier. Blocking mode is important because in many uses data may pause in between bursts, and therefore the ability to sleep is important to conserve processor resources.

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. 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 vary by platform, but for x86 remote writes are guaranteed to be seen in program order and thus store fences are inexpensive.

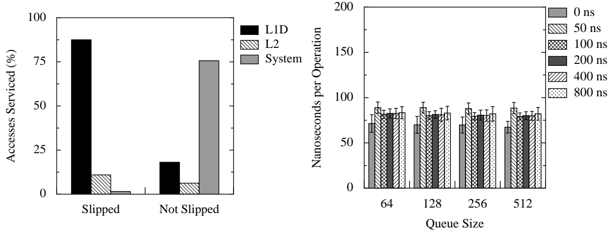


(a) FastForward with fence (spin) (b) Lamport with fence (spin)



(c) FastForward with fence (block) (d) Lamport with fence (block)

Figure 8. Queue Comparison with Fences.



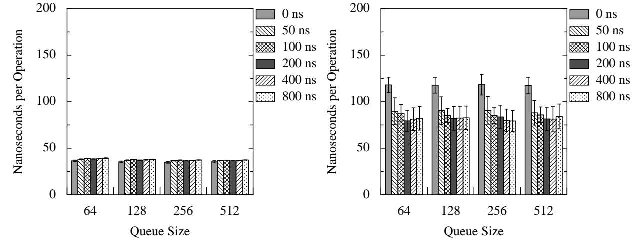
(a) Cache Performance (b) FastForward (spin, no slip)

Figure 9. Cache Behavior

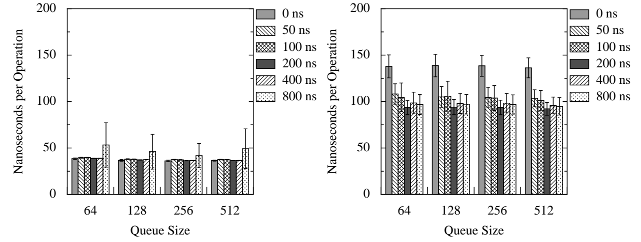
5.4 Cache Behavior

Figure 9(a) 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). To highlight how slipping behaves, we present the cache measurements for a good run and a run with deliberately unbalanced stages to cause cache thrashing.

From Figure 9(b), the difference between slipped and non slipped is immediately obvious. In the slipped case, 87.5% of the queue data references are serviced by the L1 data cache, corresponding to the predicted 8:1 reduction in compulsory misses. Further notice that 87.5% of the L1 data cache misses are serviced by the L2 cache, leaving only 1.5% of the original access to be serviced by the system. This clearly demonstrates that combining slipping with the hardware prefetcher may increase the cache hit rate to to 98.5%. As expected the non-slipped case performs poorly.



(a) FastForward (on die) (b) Lamport (on die)



(c) FastForward (off die) (d) Lamport (off die)

Figure 10. Queue Comparison, On vs. Off Die

5.5 Performance On and Off Die

This section isolates and evaluates the performance of FastForward for two stages communicating on and off die. Figure 10 shows that the performance of FastForward and Lamport are insensitive to core placement. This behavior can also be observed in the previous Figures (7 & 8), although the performance was not isolated. That FastForward is insensitive to core placement is not surprising as Figure 9 shows that most communication overhead is masked by the prefetcher. That Lamport's queue is insensitive is surprising and suggests that our AMD processors may not be taking advantage of a fast-path short circuit optimization for cache sharing enabled by the MOESI coherence protocol.

5.6 Network Frame Processing

We conclude our evaluation by confirming that the performance isolating benchmarks described above are realistic for implementation in applications by evaluating the network frame forwarding example described in Section 2.2. The time available for processing by the input and application stages (in order) is quantified in Figure 11 showing that the stage-to-stage communication time is small and constant regardless of frame size. The input time for the input stage my appear longer than expected because it includes the time to pull data out of the data structures shared with the network interface. Recall that the system is capable of forwarding a record-breaking 1.428 million 64byte frames per second with pipeline stages distributed between the kernel and user-space. Additionally, the system is capable of

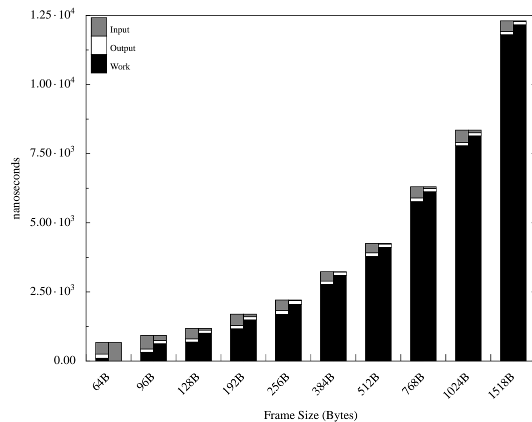


Figure 11. Network Frame Forwarding

forwarding the theoretic maximum number of frames for all frame sizes ≥80bytes. The general applicability of the technique even for small data elements is therefore confirmed. This performance is not possible with Lamport's queue.

6 Conclusion

This paper presented FastForward, a high-rate core-to-core communication algorithm for instantiating efficient fine grain pipeline-parallel applications. FastForward uses a software-only domain-specific adaptation of single-producer/single-consumer concurrent lock-free (CLF) queues to provide fully decoupled operation for cross-domain (process) communication. By leveraging temporal slipping and hardware cache prefetching, FastForward can enqueue or dequeue pointer sized data elements in 36-40 ns on the tested hardware, 2.5x faster than the next fastest CLF queue implementation and up to 4x faster for very fine grain stages (≤200 ns). The performance was found to be insensitive with respect to work load, queue size, or core allocation (i.e., on or off die). 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.

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 *International Symposium on Code Generation and Optimization*, San Jose, California, Mar. 2005.

[3] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *10th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, New York, NY, USA, 2002. ACM Press.

[4] 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*, December 2004.

[5] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 316–326, 1991.

[6] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[7] IA-32 Intel architecture software developer's manual volume 3: System programming guide. `http://developer.intel.com/design/pentium4manuals/`.

[8] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[9] M. Martin, D. Sorin, H. Cain, M. Hill, and M. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *In 34th Annual International Symposium on Microarchitecture (MICRO-34)*, December 2001.

[10] 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.

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

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

[13] 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.

[14] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, pages 177–188, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[15] 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.

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