FastForward for Concurrent Threaded Pipelines

John Giacomoni, Manish Vachharajani and Tipp Moseley
University of Colorado at Boulder

University of Colorado at Boulder
Technical Report CU-CS-1023-07 -
January 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 Concurrent Threaded Pipelines

John Giacomoni, Manish Vachharajani and Tipp Moseley
University of Colorado at Boulder

Abstract

The performance, cost, and flexibility of commodity multi-core systems make them appealing for threaded applications. Unfortunately, popular threading techniques require independent code regions, use expensive synchronization primitives, and use expensive communication mechanisms. Recently, researchers have proposed several Concurrent Threaded Pipeline architectures (CTP) which relax the data independence requirement and can increase computational throughput proportionately to the pipeline depth. Examples include Decoupled Software Pipelining, which focuses on compiler based extraction of pipelines from sequential codes, and the Frame Shared Memory architecture, which focuses specifically on network processing. CTP architectures show great promise for threading applications given a low-overhead high-speed blocking queue implementation.

This paper presents the FastForward system, a novel software-only low-overhead high-speed blocking queue implementation for CTPs. FastForward uses a novel domain-specific adaptation of concurrent lock-free queues (CLF) in conjunction with a clever memory organization to provide the fast, low-overhead, queue operations. The key to FastForward's success is its domain specific optimization based on careful tuning for modern multi-core microarchitectures. Enqueue and dequeue times are as low as 35 ns, 5 times faster than the next best solution.

1 Introduction

Traditionally, increases in transistors and fabrication technology have led to increased performance. However, these techniques are showing diminishing returns due to limitations arising from power consumption, design complexity, and wire delays. In response, designers have turned to chip multiprocessors (CMPs) that incorporate multiple cores on a single die.

While CMPs are a boon to throughput driven applications such as web servers, single-threaded applications' performance remains stagnant. Furthermore, the typical approach to parallelizing

software (be it by hand or via parallelizing compiler) has been to find, extract, and run nearly independent code regions (i.e., regions between which there are few data dependences) on separate processors [16]. However, this approach is difficult for general purpose applications [2].

Recent work has shown that many applications can be parallelized by building concurrent-threaded pipelines (CTPs) [12, 16, 13, 5]. In this paradigm, computation is divided into stages where each stage is a thread and each thread runs on a separate core. Pipeline stages communicate via some communication primitive. One method, the Decoupled Software Pipelining (DSWP) approach constructs a CTP from loops in sequential programs [16]. Ottoni et al. demonstrate an automatic compiler [12] technique that can extract pipelines from a variety of benchmarks and delivers 9.2% mean performance improvements.

Another example of a CTP is the Frame Shared Memory Architecture (FShm) network processing framework [5]. It is built by pipelining the kernel network I/O layers along with the user-space frame processing logic, yet it still maintaining traditional OS safety guarantees. Using only commodity hardware, the FShm CTP is able to process network frames at Gigabit Ethernet *throughput* rates (as often as every 672 ns) even though per-frame processing *latency* easily exceeds 672ns on the same hardware [5].

Clearly, CTPs are a promising mechanism to parallelize applications for performance improvement, even for general-purpose applications. However, they require a buffering communication mechanism that (1) provides fast enqueue and dequeue operations (less than 50ns [12]), (2) has no synchronization unless the reader and writer are operating on the same queue slot, (3) has a small resource footprint so as not to interfere with application logic, and (4) supports common OS functionality such as process migration. Unfortunately, these features are difficult to deliver on commodity hardware. As evidence, consider that DSWP relies on a custom core-to-core hardware communication mechanism, the synchronization array [16], whose implementation requires significant modifications to the processor pipeline.

In this work, we present FastForward, a novel *software only* buffering communication mechanism for CTPs running on commodity multi-core hardware. FastForward provides fast queue/dequeue operations by using a shared-memory region to construct CTP specific concurrent lock-free queues [10]

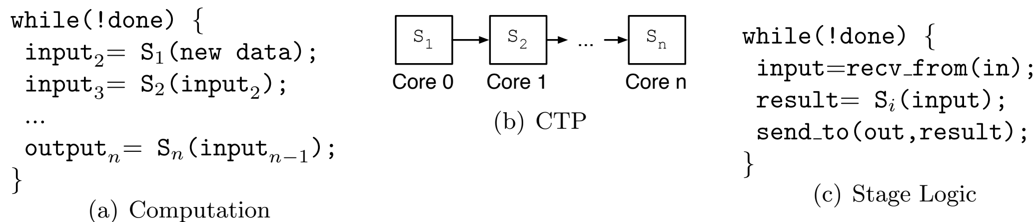


Figure 1: A Simple Concurrent, Threaded Pipeline (CTP)

that eliminate all synchronization operations. FastForward relies primarily on the store atomicity guarantees of the cache coherence mechanism to provide blocking semantics and avoid data races. For more performance, FastForward uses a clever memory organization and a CTP-specific timing strategy to allow the prefetcher to eliminate almost all coherence misses. Furthermore, since all FastForward state is stored in memory, OS process migration is handled seamlessly. FastForward can also support cross-protection-domain (e.g., between kernel and user-processes) communication by allowing user and kernel-space to share memory. Experiments on a dual-processor, dual-core AMD Opteron 270 (2 GHz) show that FastForward permits queuing and dequeuing in approximately 35 ns, 5 times faster than the next best solution.

The remainder of this paper is organized as follows. Section 2 gives background on concurrent-threaded-pipelines (CTPs) and develops the requirements for FastForward. Section 3 describes the implementation and tuning of FastForward. Section 4 discusses advanced implementation issues with FastForward. Section 5 presents an evaluation of FastForward on multi-core hardware. Section 6 discusses prior work. Section 7 concludes.

2 Concurrent Threaded Pipelining

The FastForward communication mechanism presented in this paper is specifically designed to enable high-performance CTPs, such as DSWP and FShm, on commodity hardware. Other examples are briefly discussed in Section 6. This section gives an overview of CTPs to develop a set of feature requirements and performance targets for FastForward. The discussion begins with a simple example and proceeds to more sophisticated pipelines.

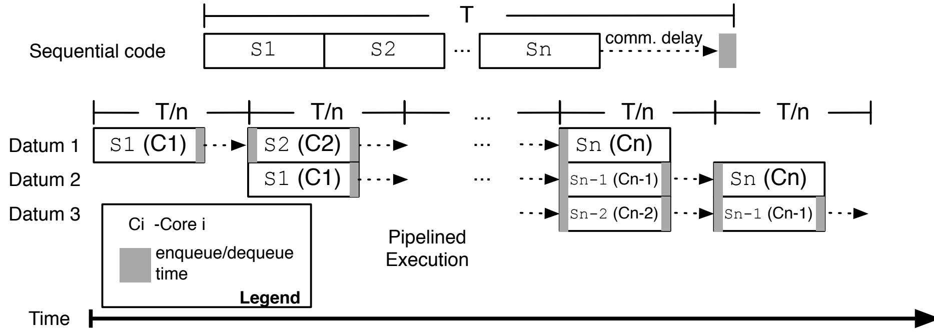


Figure 2: Timing Diagram for a Simple CTP

2.1 A Simple CTP

A CTP is a software organization that allows data dependent computations to be parallelized on multi-core platforms. Consider the hypothetical computation shown in Figure 1a. Assume that each function `S1`...`Sn` is functional (i.e., its return value depends only on its input, there is no internal state). From this code, we can build the simplest of CTPs shown in Figure 1b. Each processor runs a pipeline stage that repeatedly executes one of the `Si` functions on each new input it receives. The computation done in a stage i is shown in Figure 1c.

An idealized execution of this pipeline is illustrated in Figure 2. Here, we assume that each stage takes exactly T/n time units to execute and that the communication delay is small. Note that although there are data-dependences between any two consecutive `Si`, computation still proceeds in parallel with each processor completing useful work at any given time. This is the key benefit of concurrent-threaded pipelines; code with data-dependences can still be parallelized for performance gains.

2.2 Decoupled Software Pipelines

Unfortunately, most real code sequences do not follow the assumptions in the simple example above. First, it is likely that each stage will have variable timing due to cache misses, input variations, etc. Furthermore, most commodity hardware does not provide very short core-to-core communication latencies (although latencies are shorter than on older SMP systems).

Fortunately, CTPs can still extract performance from real-world codes by building a DSWP, a

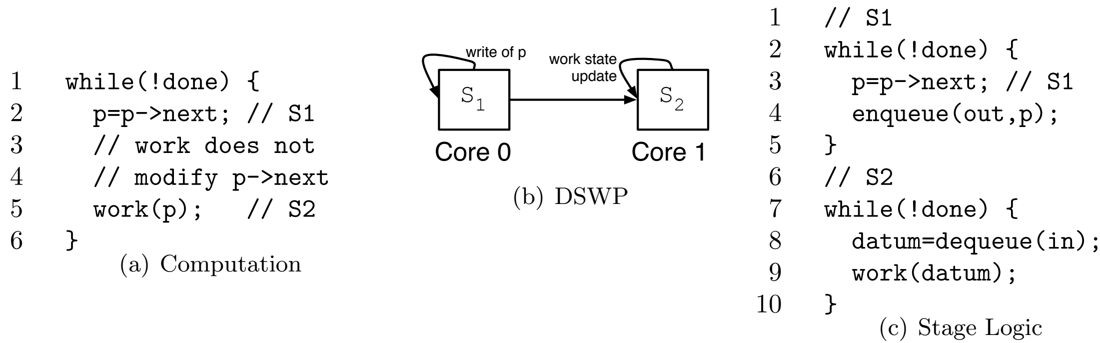


Figure 3: A Simple Decoupled Software Pipeline (DSWP)

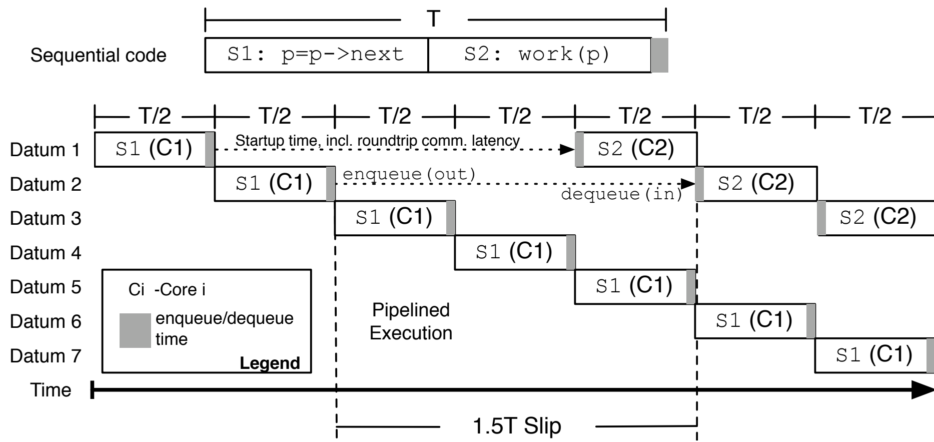


Figure 4: Timing of a Slipped DSWP

type of CTP. In DSWP, each pipeline stage computes using only its inputs, updates only stage-local pipeline state, and as output provides only the inputs of later pipeline stages.

Figure 3a is an example of a simple computation that can be parallelized into a DSWP (originally presented in [16]). Here, we can consider the `p=p->next` code as one stage in the pipeline and the `work(p)` as another, as shown in Figure 3b and c. Notice that here, stage `S1` generates the input to `S2` using only its internal state (in particular, the value of `p` and `p->next`). `work` generates its result (as side-effects on memory) using only its input and `work`-local state. Notice how DSWPs can have stateful pipeline stages and thus are able to handle more complex dataflow patterns, in particular those involving select recurrences due to loop backedges [16].

Because pipeline stages in DSWP only operate on local state, data only flows in one direction between stages as shown in Figure 3b. As a result, pipeline stages are decoupled, which means

that communication latencies can be removed from the critical path and rolled into the pipeline startup and tear-down cost which is amortized over all computation. This is shown in the timing diagram in Figure 4. Here, we still assume that stages consume a fixed amount of time, but, unlike the simple CTP, in this figure, long communication latencies have been introduced, but only two end-to-end communication latencies are on the critical path, one for startup, shown in the figure, and one for tear-down (not shown).

Figure 4 also shows that long latency communication can be absorbed by *slipping* the relative execution times of stages; earlier pipeline stages are allowed to execute well ahead of stages that consume their results. To support this slipping, however, the underlying communication mechanism must support buffering of earlier data produced by a pipeline stage so that later executions do not overwrite values before they can be consumed by the next stage. An in-order queue will suffice for this purpose. Note, however, that the time taken to enqueue and dequeue operations from this queue *is* on the critical computation path, even though the end-to-end latency is not. Therefore, the enqueue and dequeue operations must be very fast. Ottoni et al. suggest no more than a 100 clock cycles, or 50 ns on a 2 GHz machine [12]. This result generalizes to all CTPs including DSWPs.

One benefit of this ability to slip relative processing of data in CTP stages is that variation in per-stage processing time is easily tolerated. Delays in an earlier stage reduce the amount of slip between it and later stages. Delays in a later stage increase the slip between it and earlier ones. The dynamic nature of this slipping yields considerable benefit; it allows DSWPs to overlap useful work with cache miss penalties without requiring huge instruction windows [16, 18]

An exciting feature of DSWPs is that Ottoni et al. have built a compiler that extracts them automatically from SPEC and other benchmarks [12, 13]. Although they require *custom* communication hardware that is integrated into the processor pipeline to gain performance [16], technologies like FastForward along with the DSWP compiler may enable automatic parallelization of sequential codes on *commodity* multi-core systems.

2.3 More General CTPs

In general, one can think of CTPs as the asynchronous software analog of traditional synchronous processor pipelines. Each CTP stage is equivalent to one stage in a processor pipeline. The CTP queue buffers are analogous to the pipeline registers in a processor. Interlocks in the processor pipeline (to handle data and control hazards) are implemented in a CTP by stalling a stage if its input buffers are empty (implying blocking queue semantics for the communication mechanism). Communication and state update within a stage, but across different inputs, is analogous to forwarding from a pipeline stage, A, back to stage A. Thus, a DSWP is analogous to a processor that only needs to forward from one stage back to that same stage (e.g., only EX to EX and Mem to Mem forwards).

Modern pipelined processors also support forwarding from a later stage to an earlier stage (e.g., the Mem to EX forward in a 5-stage pipeline [15]). Thus, general CTPs can also be constructed with such communication paths and a buffering queue with fast enqueue and dequeue operations suffices for this purpose, just as before. However, note that this type of communication can be problematic in a CTP (and are thus avoided in DSWPs) because they couple the relative timing of pipeline stages, thus reducing the ability of the CTP to slip. Despite this, FastForward supports these communication paths naturally.

2.4 Required Communication Features

With an understanding of CTP basics, we can now outline the requirements for FastForward, our novel software-only CTP communication structure for commodity multi-core hardware. The requirements are as follows. (1) As discussed above, FastForward must support in-order enqueuing and dequeuing of data. (2) Since stalls are created by blocking on empty or full queues (as discussed above), the queues must have blocking semantics from the application perspective. The block can be done by spin-waiting or true sleep depending on system needs (see Section 3). (3) Since the enqueue and dequeue latency is on the critical path, these operations should be fast (ideally less than 50ns [12]).

In addition, a bit more consideration of CTPs and their interaction with other system compo-

nents, such as the operating system (OS), yield these additional requirements. (4) FastForward's resource footprint should be small so as not to interfere with actual computation. (5) FastForward should be able to ensure correct semantics even when the OS migrates processes from core to core. (6) FastForward should help ensure that pipeline stages with real work to complete are scheduled ahead of those that are stalled waiting for input. (7) FastForward should export a portable interface that allows for future hardware accelerated communication and transparent incorporation of pipeline stages built from custom hardware blocks (e.g., a TCP on-load engine [17]).

For robustness, FastForward should also (8) support conditional non-blocking reads of select queues and (9) allow out-of-order processing of data if requested. Out-of-order packet processing can help a CTP avoid stalls due to certain data or structural hazards. Conditional non-blocking reads on certain queues allows CTPs to support asynchronous "interrupts" (to continue the earlier analogy). This type of interrupt is common in many applications, especially those where output quality may vary. Consider a CTP implementing a video decoder. The conditional channel can indicate that the viewer wishes to fast-forward or skip to another location in the video.

In the next section, we see that FastForward is able to deliver all these features using only commodity hardware.

3 FastForward

This section presents FastForward. In particular, it describe how CTP-specific knowledge permits a high-performance communication primitive ($\approx 35ns$ for enqueue/dequeue) in software. The section begins by examining the bottlenecks in traditional lock-based queues, and then proceeds to describe how FastForward overcomes these bottlenecks. Finally, we describe how to tune FastForward to maximize performance on modern memory subsystems.

3.1 Traditional Lock Based Queues

Recall that the CTPs require a fast buffering communication primitive capable of providing blocking semantics and that queues are a natural fit. However, queues implemented with standard synchro-

Lib	Lock	Unlock	Total (ns)
pthread	75	78	159
kernel	28	19	27

(a)

# VCPU	# Thr.	Syscalls (ns)	ioctl (ns)
0	1	171	538
1	1	851	1826
1	2	1334	1674
2	2	911	1284
2	3	912	1223

(b)

Figure 5: Mutex operation costs (a) and system-call, and ioctl costs (b) for an AMD Opteron. Note: For VCPU=0, no scheduler activations are used; scheduler activations result in reduced performance even for single thread tests.

nization primitives are too slow to meet the performance requirements of CTPs (as outlined in [12]).

Recall that traditional queues lock the queue structure during enqueue and dequeue (see Figure 6). Figure 5a shows the cost of these high-level mutex operations as provided in the kernel and by the native libpthread library on an Opteron 270 with FreeBSD 5.5. Notice that the native libpthread library performs 3x worse than the $50ns$ upper bound on queue operation performance for CTPs set by Ottoni et al. [12], all without actually performing any queue operations. Figure 5b demonstrates that if system calls are needed, as with cross-domain communication in FShm, matters become worse. A single system call costs at least 4.5x more than Ottoni's upper bound. If the transferring process has threads running the cost is at least 18x greater than the bound. Batch transfers can amortize these costs, but FastForward can dramatically reduce the per-transfer operation making this unnecessary. Furthermore, FastForward requires no high-level operating system primitives and thus the queue/enqueue time is fixed regardless of transfer type.

3.2 Classic CLF Queues

In order to provide a fast, synchronization-primitive-free, communication mechanism, FastForward builds on the framework of concurrent lock-free (CLF) queues. CLF queues, when properly implemented, support the critical property necessary for performance in a CTP; they are wait-free [11], implying that sends are non-blocking and that system is starvation free.

While CLF queues have been extensively studied, most variants are based on linked-lists and

```
1   enqueue(data) {
2     lock(queue);
3     // Queue full
4     while(NEXT(head) == tail) {}
5     buffer[head] = data;
6     head = NEXT(head);
7     unlock(queue);
8   }
```
(a) Enqueue

```
1   dequeue(data) {
2     lock(queue);
3     // Queue empty
4     while(head == tail) {}
5     data = buffer[tail];
6     tail = NEXT(tail);
7     unlock(queue);
8   }
```
(b) Dequeue

Figure 6: Traditional Lock-based Queue Operations

```
1   enqueue(data) {
2     h = head;
3     while(NEXT(h) == tail) {}
4     buffer[h] = data;
5     h = NEXT(head);
6   }
```
(a) Enqueue

```
1   dequeue(data) {
2     t = tail;
3     while(head == t) {}
4     data = buffer[t];
5     tail = NEXT(t);
6   }
```
(b) Dequeue

Figure 7: Traditional Concurrent Lock-free Queues

therefore unacceptable as FastForward must avoid potentially long-latency cache misses and memory allocation costs. Array-based alternatives [9, 8] exist, however in the general case, they require a double-word atomic compare-and-set instruction or load-linked and store-conditional [11].

Fortunately, in the CTP domain, single-producer to single-consumer (SPSC) queues sufficient. Thus, Lamport or Massalin and Pu's [9, 8] lock-free code (summarized in Figure 7) suffice on sequentially consistent hardware [8].

3.3 FastForward's CLF Queues

One of the main implementation challenges in FastForward was to make this implementation work on a modern cache-coherent Non-uniform Memory Architecture. In particular, since these machines have non-sequentially consistent memory models [4], operations have been carefully ordered so that proper queue semantics are guaranteed via only cache coherence guarantees (in particular store atomicity).

```
1   put_nonblock(...) {
2     if (NULL == queue[head])
3     {
4       queue[head] = ptr;
5       head = NEXT(head);
6     }
7   }
```

(a) Enqueue

```
1   get_nonblock(...) {
2     if (NULL != queue[tail])
3     {
4       ptr = queue[tail];
5       queue[tail] = NULL;
6       tail = NEXT(tail);
7     }
8   }
```

(b) Dequeue

Figure 8: FastForward Queue Operations (non-blocking for clarity).

Figure 8 shows pseudo-code for the non-blocking put and get operations. Careful reading shows that for each operation only two memory locations are accessed, the array slot and the index into the array. Massalin & Pu and Lamport read and compare the head and tail indices to check for queue full/empty conditions. As we will see, this is a critical optimization.

First, the optimization permits CTPs to keep the cache lines containing the indices in the modified (M) state [14] when there is no data to read or no space to write. In the traditional code (Figure 7), when there is no need to spin-wait, the cache line for the head and tail indices will thrash between the M and the S states because of the spin check on line 3 for queue and dequeue respectively. This can be expensive.

The cache line coherence protocol state transitions are eliminated on a successful queue operation by directly accessing the queue buffer and eliminating the head/tail comparison. With this scheme, two sequential writes or reads to the queue by the same CTP stage (i.e., thread) result in a cache hit for the second access. In the classic CLF code there would still likely be a cache miss on the spin check. The caveat with this optimization is that there must be a well defined value signaling *empty*. With pointers as the common CTP queue payload, `NULL` suffices. With other payloads, one must ensure that there is at least one non-zero field.

Finally, notice that this optimization also allows a CLF implementation that only depends on the coherence protocol (and not the consistency model) to ensure mutual exclusion (see Section 4 for caveats with pointers to payload data). To see this, recall that stores are atomic on all coherent memory systems, thus the getter and putter will never update the same node simultaneously since

the getter waits for a non-`NULL` value and the setter a `NULL` value.

3.3.1 Reducing Memory Hotspots

Avoiding thrashing of cache lines (memory hotspot) is critical for FastForward (and CLF queues in general). Furthermore, they can be difficult to manage. Consider a CTP where threads (i.e., stages) are separated in time by a single enqueued item (as in Figure 2). With the FastForward implementation described thus far, if the buffers processed sequentially every enqueue and dequeue will cause the cache line to move between cores.

The natural optimizations are to loosely pack the array so that only one element is stored per cache line. This solution attempts to reduce contention by having the full/empty case access a new cache line. However, every successful enqueue or dequeue operation still requires the coherence protocol to move a cache line from one processor to another, thus the hotspot is still present in the common case for CTPs.

FastForward eliminates these by leveraging the fact that a CTP's sole goal is to maximize throughput through the pipeline of asynchronous stages. If we accept a small increase in latency it is possible to reach the theoretic minimum cache line thrashing by slipping the processing stages in time (see Figure 4 so that an entire cache line worth of data can be successfully processed with no coherence work. For example, for a machine with 64 Byte cache lines, and 8 Byte queue slots (e.g., a 64-bit pointer) one can pack 8 queue entries per cache line. Thus, if the CTP is slipped by 8 stages, the writing stage will fill a cache-line before the reader begins to read it. Thus, for every sequence of 8 operations there is only a single compulsory miss. Since the stages are separated by at least a cache line, they can continue to stream in what previously would have been a line thrashing situation.

3.4 Maintaining Slip Despite Stage Variation

To this point, all solutions have assumed that the software CTP stages have fixed execution intervals like synchronous hardware pipeline stages. Unfortunately, this is not the case; interrupts, scheduler activity, and cache misses cause variation (i.e., jitter) in the stage timing. Positive jitter extends

the amount of slip in the pipeline making cache line trashing less likely, though it causes slight reduction in throughput in the no-thrash case. Negative jitter reduces slip and can cause trashing, but is critical to certain types of CTPs for performance. For example in DSWP [16, 12] positive jitter is used to create tolerance for negative jitter due to cache misses and is a source of significant performance improvement.

FastForward provides a primitive that periodically polls the upstream stage to ascertain the amount of slip in terms of cache lines. If there is insufficient slip to prevent thrashing, stalls are inserted into the pipeline so as to introduce positive jitter and eliminate thrashing. As we will see in Section 5.4, this dynamic adaptation along with slip are the key reasons for FastForward's performance.

Short running threads are extremely damaging as they will erode any temporal slipping, causing hotspots as cache lines are bounced between the threads. Further, a short running thread will effectively thrash the coherence manager as it constantly polls the queue for work interfering with the producer's updates. Fortunately this problem is easily solved by introducing artificial work at the end of each stage when it runs for less than the average running time of the longest stage. By spinning on a timestamp it is possible to spin only in negative situations and quickly skip past in the positive case (In our tests we spin on the TimeStampCounter register). Notice how spinning will have no overall impact on throughput as a pipeline's throughput is defined by the longest running stage.

Section 4 outlines how the hardware can help hide the per cache line compulsory work and how more advanced memory managers can reduce the latency caused by temporally slipping the threads.

3.4.1 Application Overhead

With any software or hardware construct the implementation's footprint may be a concern. Specifically there is a danger in polluting both the instruction and data caches if the queue logic is overly complex or the buffering storage requirements are high. As described above, our queues use minimal logic per operation and minimize wasted space by densely packing datum into each cache line.

Section 5 demonstrates excellent performance can be achieve with small queues.

4 FastForward's Advanced Features

This section discusses some advanced features of FastForward that were deferred in the previous section.

4.1 Large Payloads

Section 3 assumed that each data element in the FastForward queue could be read and written with a single atomic instruction. Unfortunately, data elements are often larger than can be written atomically (e.g., network frames, and matrix sub-blocks). Payloads larger than a single word, but smaller than a cache line can still be inserted directly into the queue by writing each word separately and using the final word as the synchronization trigger. Blocks too large for this approach can be supported by inserting a reference to an external data block into the queue. FShm uses this technique and arranges the pipeline as a ring so empty buffers are recycled automatically.

Note, however, that using a reference to payload data introduces a dependence on the processor consistency model for correctness. On weakly-consistent machines, writes to the queue slot can be re-ordered with respect write of the payload data. Thus, the CTP stage on the receiving end of the queue may dequeue an element from the channel *before* the payload data is visible to it. Thus the reader could subsequently read stale data from the payload memory. On processors that guarantee that stores generated by a given processor are seen in program order by other processors, this cannot occur. The x86 memory consistency model makes this guarantee [1]. On processors without this guarantee, a store-fence before the enqueue of the payload will cause all prior payload writes to commit and avoid this problem. Platforms that guarantee store order in effect insert a store fence after every store, so performance should not be impacted.

4.2 Cross-Domain

FastForward supports CTP stages from a privileged domain, such as the kernel, within a user-space application. Thus, it is possible to have a stage that performs file and network I/O [5] without system-call overhead. FastForward supports cross-domain communication by creating a shared memory region that joins the different domains. The CLF queue is built in this region. Since the FastForward queues require no high-level synchronization between participants, no modifications are necessary to the basic queue routines. Full blocking semantics do require a shared condition variable.

4.3 Transparent Portability

The FastForward is portable and polymorphic. It permits the underlying queue implementation to be changed without modifications to the application. An efficient abstraction layer is implemented using function pointers since the hardware jump target predictor removes much of their overhead. Situations in which this minor overhead is unacceptable, can access the queues directly with inline functions. FastForward currently uses the polymorphic interface to different queue blocking semantics as well as a traditional heavyweight serialized queue supporting cross-domain communication. One important aspect of this polymorphism is that the underlying queue implementation can be adapted on-the-fly to resource demands; spin-waiting can be used when there are more processors than ready threads, and sleep-based blocking can be activated when the system is over-subscribed. Finally, the polymorphic interface can be used to dynamically plug-in hardware accelerated implementations (e.g., based on the synchronization array [16]) or to communicate with stages implemented in hardware (e.g., a TCP on-load engine [17]).

4.4 Advanced Architectural Interactions

Our experimental evaluation showed two interesting properties of the Opteron system that provide considerable performance benefits. First, because of the L2 stride prefetcher, only 0.75% of all L1D cache accesses are serviced by main memory or cache-to-cache transfers when stages are slipped. Given the performance of the stride prefetcher, a content prefetch unit [3] would help hide the cost

of accessing a pointer referenced data payload. Transactional Memory [6] may be able to mask the cache coherence costs when two threads are sharing a cache line or a thread prematurely attempts to access the next entry.

5 Evaluation

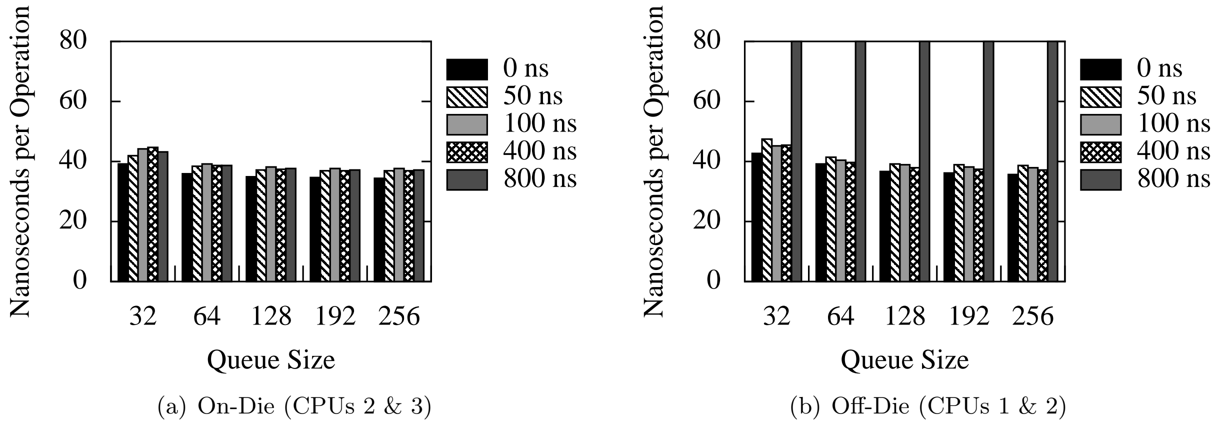
This section presents an evaluation of FastForward. It shows that FastForward's performance is work load invariant and queue size invariant. Furthermore, the performance of communicating on and off die, the ideal amount of temporal slipping, and underlying cache behavior are measured.

Section 5.3 compares the performance of the FastForward across both queue size and varying stage workloads both, with communication both on and off die. Section 5.4 shows the cache behavior of SPSC CLF queues when they are temporally slipped and when they begin chasing cache lines with little or no slip. Results confirm analysis from Section 3.3 that predict how traditional CLF queues would behave due to the head/tail comparisons.

5.1 Software Setup

Each of these parameters is evaluated on looped pipelines with 2 or 3 stages [1] in which the last pipeline stage communicates a message back to the first. We use a looped pipeline to demonstrate message passing with pointer-based external payload buffers. The pipeline is such that a reference to a single payload buffer is passed through each stage and then recycled back to the producer. 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 the cycle accurate TimeStampCounter register available on the test hardware. In all executions one million frames were passed through the pipeline and the average per frame throughput rate computed. Given the throughput rate, the queuing costs can be identified by subtracting the time spent spinning.

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



(a) On-Die (CPUs 2 & 3)

(b) Off-Die (CPUs 1 & 2)

Figure 9: Spin vs. Queue Size, 2 stage loop

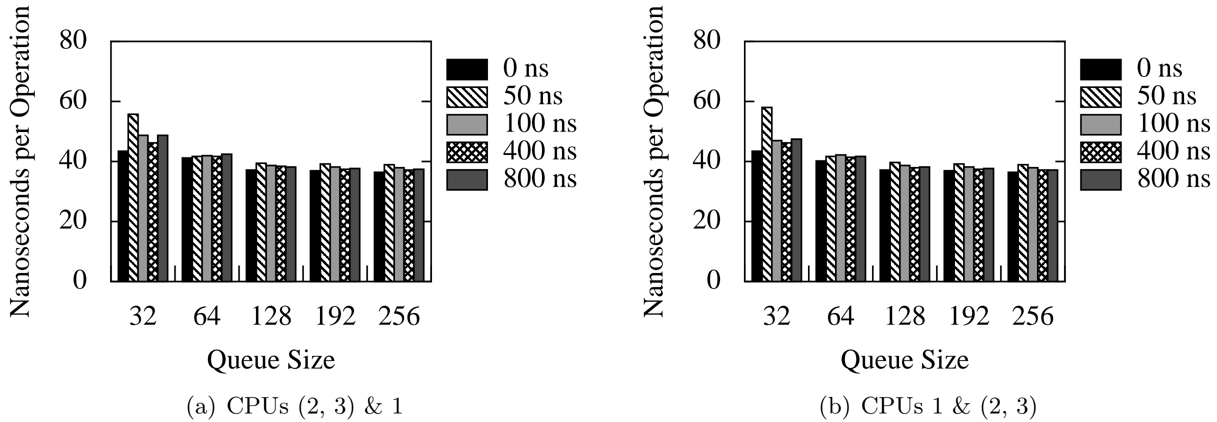
As a point of comparison we also implemented and measured a traditional lock-based queue on our experimental platform. As discussed below, baseline experiments revealed per operation costs starting at 400-600 ns and ran up to 20,000 ns and beyond in degenerate cases. At worst, FastForward is 10x faster than the baseline lock-based queues and up to 1000x faster in degenerate cases.

5.2 Hardware Platform

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. The relevant chipset, which can have a dramatic effect on performance, is the AMD-8111 (HyperTransport I/O Hub).

5.3 Performance

This section examines the performance of FastForward's CLF queue implementation with respect to work load, queue size and core selection (either on-the-same-die or off-die). Cache behavior and the power of temporal slipping is discussed in Section 5.4 below. Figures 9 and 10 summarize results for two and three stage pipelines with different stage/core arrangements. Each group of bars shows the cost of a queue and enqueue operation for a given queue size (number of slots). Each bar in the group shows the cost of the enqueue and dequeue operations for work loads that take a varying amount of time.



(a) CPUs (2, 3) & 1

(b) CPUs 1 & (2, 3)

Figure 10: Spin vs. Queue Size, 3 stage loop

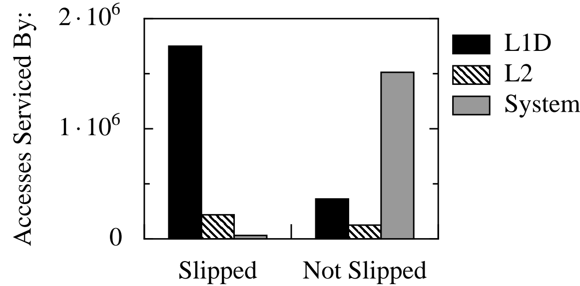
The main observation is that FastForward's performance is insensitive to both the queue size and simulated work load. Figure 9b shows a consistent anomaly with a workload of 800 ns that is mirrored in our locking queue (baseline). The baseline implementation takes over 20,000 ns in this case. Note that the three-stage pipeline does not suffer from this aberration. We believe this to be due to an unknown pathological interaction with the underlying hardware or OS resulting in loss of temporal slip. We believe that this difficulty can eventually be remedied. Observe that the system scales cleanly from two to three stages and should continue to scale as additional cores are added.

Figures 9b and 10b shows the performance of cross-die communication to be, non-intuitively, equivalent to on die communication. Recall, however, that FastForward maintains temporal slip, therefore the hardware has the ability to prefetch the subsequent cache line and thus hide the cross-die communication costs (Section 5.4).

5.4 Cache Behavior

To compare the extremes of cache behavior for our queues we instrumented the code and measured the number of level 1 data cache accesses and how many of the misses were serviced by the level 2 cache or by the system (both cache-cache transfers and transfers from main memory). We took the best on-die run and the worst run from the data in Figure 9.

Notice how in the best performance case FastForward has fully slipped the pipeline, as evidenced



System denotes both cache-cache sharing and main memory accesses.

Figure 11: Cache Performance

by the high number of level 1 data cache hits (87.5% of queue accesses). Further notice how the bulk of the L1D misses are serviced by the level 2 cache. Since 1/8th of the L1D misses must be compulsory or coherence misses (see Section 3), this indicates that misses are addressed by the prefetcher. Examining the worst case, we clearly see that not only are the stages not slipped, but the prefetcher is certainly not active.

To this point, we have compared FastForward to traditional lock-based queues. However, a true measure of FastForward is the performance advantage it delivers over CLF queues (see Section 3). The performance of these queues can be conservatively approximated by running FastForward without slipping [2]. Without slipping, we measure an average enqueue/dequeue cost of approximately 200 ns. Thus, FastForward is at least 5x faster (35 ns vs. 200 ns) than the best comparable software-only communication mechanism.

A comparsion of the enqueue/dequeue cost of FastForward, traditional lock-based queues, and concurrent lock-free queues is shown in Figure 12.

6 Prior Work

The prior work upon which FastForward is based is discussed throughout the paper. This section elaborates on some of that work and discusses a few other related works.

[2]The measurement is conservative because it gives CLF queues the benefit of FastForward's head-tail optimization.

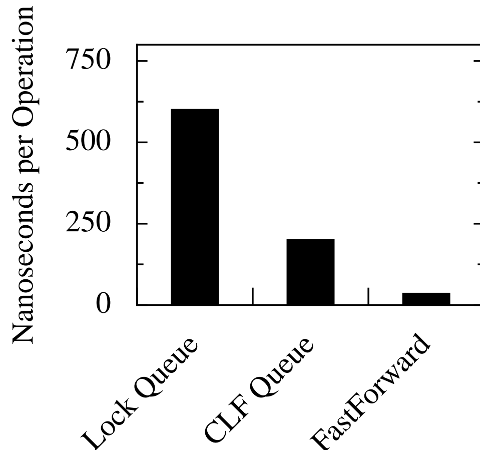


Figure 12: Summary comparison of FastForward to CLF queues and lock-based queues.

6.1 Architecture Based Communication

Early work [18] by Smith et. al. focused on decoupling memory access from execution, providing the foundation for implementing pipelines in software [16, 12, 13, 5]. Note that in these previous efforts, stages were designed to work in lock-step thus causing both stages to be a single schedulable entity. Should either stage lag or be paused, both stages were implicitly paused. Ideally any stage should be able to proceed provided it has input, otherwise any time a stage lags due to memory accesses, a processor takes an interrupt, or a stage is temporarily halted by the scheduler, the entire pipeline stalls.

More recently, Rangan et. al. proposed the idea of Decoupled Software Pipelining (DSWP) [16]. Recognizing the difficulty in correctly scheduling stages to prevent undesirable stalls to the entire pipeline, they proposed decoupling the stages by eliminating circular dependencies. Stalls are automatically accounted for by using low-cost (in terms of cycle count) *hardware* queues to connect stages and thus provide automatic buffering of messages. Further Ottoni et al. demonstrated that much of this parallelization work can be accomplished automatically by a compiler [12].

6.2 Software Organization

Early work on Software Pipelining (SWP) focused on rearranging loops to create an instruction pipeline [7]. However, despite similarities in the name, SWP is designed to expose ILP for a single core. Thus it is complementary to CTPs in that it can be used to optimize stage logic.

The Synthesis system [9] provided the tools necessary to build a pipeline in software, but the focus was on building the most efficient message-passing system possible [9]. Unlike previous systems, Synthesis used concurrent lock-free queues to provide the communication channel, used the notion of asynchronous communication, and applied thread synthesis (fusing two threads together and removing the communication queue) when the cost of communication was observed, at runtime, to be overly expensive. FastForward builds heavily on this work as discussed in Section 3.

The Frame-Shared-memory architecture also built on this work but added concurrency as a requirement and demonstrated the increased throughput of a pipeline for network frame processing [5]. Additionally FShm demonstrated the power of cross-protection-domain pipelines by linking kernel and user-space threads to form a single network frame processing application.

7 Conclusion

In the transition to multi-core architectures, a method to allow applications to fully leverage these systems is vital. Recent research in various Concurrent-threaded Pipeline software architectures such as DSWP and FShm show great promise, but require a low cost buffering communication mechanism to deliver performance improvements. However, memory subsystem behavior makes delivering such a system difficult.

This paper describes the FastForward queuing system for CTPs. FastForward optimizes concurrent lock-free queues by carefully considering the interactions with the cache coherence mechanism and CTP behavior. By slipping the relative timing of execution in CTP stages, FastForward allows stages to communicate with very low overhead. Enqueue and Dequeue operations complete in approximately 35ns, over 5 times faster than the next best solution.

FastForward, combined with technologies such as FShm or the DSWP compiler (which automat-

ically extracts CTPs for multicore systems from sequential programs) paves the way for continued performance improvements as manufacturers deliver systems with increasing numbers of cores.

References

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

[2] Amarasinghe, S. Multicores from the compiler's perspective: A blessing or a curse? In *2005 International Symposium on Code Generation and Optimization (CGO)*.

[3] Cooksey, R., Jourdan, S., and Grunwald, D. A stateless, content-directed data prefetching mechanism.

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

[5] Giacomoni, J., Bennett, J. K., Carzaniga, A., Vachharajani, M., and Wolf, A. L. Fshm: High-rate frame manipulation in kernel and user-space. Tech. rep., Univerity of Colorado at Boulder, 2006.

[6] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of theTwentiethAnnual International Symposium on Computer Architecture* (1993).

[7] Lam, M. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (New York, NY, USA, 1988), ACM Press, pp. 318–328.

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

[9] Massalin, H., and Pu, C. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)* (1989), vol. 23, pp. 191–201.

[10] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1996), ACM Press, pp. 267–275.

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

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

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

[14] Papamarcos, M. S., and Patel, J. H. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture* (New York, NY, USA, 1984), ACM Press, pp. 348–354.

[15] Patterson, D. A., and Hennessy, J. L. *Computer organization and design (2nd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

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

[17] Regnier, G., Makineni, S., Illikkal, R., Iyer, R., Minturn, D., Huggahalli, R., Newell, D., Cline, L., and Foong, A. Tcp onloading for data center servers. *Computer 37*, 11 (2004), 48–58.

[18] Smith, J. E. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst. 2*, 4 (1984), 289–308.