Support for High-Frequency Streaming in CMPs

Ram Rangan Neil Vachharajani Adam Stoler

Guilherme Ottoni David I. August George Z. N. Cai†

Departments of Computer Science and Electrical Engineering
Princeton University
Princeton NJ 08540
{ram, nvachhar, astoler, ottoni, august}@princeton.edu

† Digital Enterprise Group
Intel Corporation
Hillsboro OR 97124
george.cai@intel.com

ABSTRACT

As the industry moves toward larger-scale chip multiprocessors, the need to parallelize applications grows. High inter-thread communication delays, exacerbated by over-stressed high-latency memory subsystems and ever-increasing wire delays, require parallelization techniques to create partially or fully independent threads to improve performance. Unfortunately, developers and compilers alike often fail to find sufficient independent work of this kind.

Recently proposed *pipelined streaming* techniques have shown significant promise for both manual and automatic parallelization. These techniques have wide-scale applicability because they embrace inter-thread dependences (albeit acyclic dependences) and tolerate long-latency communication of these dependences. This paper addresses the lack of architectural support for this type of concurrency, which has blocked its adoption and hindered related language and compiler research. We observe that both manual and automatic techniques create *high-frequency* streaming threads, with communication occurring every 5 to 20 instructions. Even while easily tolerating *inter-thread* transit delays, high-frequency communication makes thread performance very sensitive to *intra-thread* delays from the repeated execution of the communication operations. Using this observation, we define the design-space and evaluate several mechanisms to find a better trade-off between performance and operating system, hardware, and design costs. From this, we find a light-weight streaming-aware enhancement to conventional memory subsystems that doubles the speed of these codes and is within 2% of the best-performing, but heavy-weight, hardware solution.

1. INTRODUCTION

Chip multiprocessors (CMPs) have emerged as the predominant organization of future microprocessors. CMPs overcome the clock speed, power, thermal, and scalability problems plaguing aggressive uniprocessor designs while continuing to provide additional computing power using additional transistors provided by technology scaling. While additional processors on the chip improve the throughput of many independent tasks, they, by themselves, do nothing to improve the performance of individual tasks. Worse still for task performance, processor manufacturers are considering using simpler cores in CMPs to improve power/performance. This trend implies that single task performance will *not* improve, and may actually degrade. Thus, performance improvement on a CMP requires that programmers or compilers parallelize individual tasks into multiple threads.

High inter-thread communication delays have made the notion of thread extraction almost synonymous with the search for long-running threads with minimal communication. While this strategy has had some success for scientific applications, it has impaired similar efforts for general-purpose applications (both manual *and* automatic). Recently however, language and compiler *streaming* techniques (StreamIt [26, 6], Decoupled Software Pipelining [21, 15], and others [3, 4]) have shown promise as viable methods to expose thread-level parallelism. They can handle more codes because they embrace inter-thread dependences (albeit acyclic dependences) by partitioning applications into concurrent, long-running producer and consumer threads. They place fewer demands on interconnect latency because they easily tolerate long-latency inter-thread communication by pipelining acyclic communication.

While streaming techniques show promise, current architectures are without sufficient architectural and operating system support for pipelined streaming. This lack of support excludes an entire class of potential multi-threaded applications, discourages future development of streaming techniques, and makes the programmer's and compiler writer's job more difficult. To address this problem, this paper contributes the following:

1. A characterization of existing pipelined streaming applications, revealing their *high-frequency* nature with communication occurring every 5 to 20 dynamic instructions.
2. A comprehensive design space characterization of communication support for high-frequency streaming applications and a detailed exploration of several interesting design points in this design space, considering performance, hardware complexity, and operating system costs.
3. The identification of a low-cost design point which provides efficient support for high-frequency streaming applications, yielding twice the performance of conventional memory subsystems with minimal design changes. This solution is within 2% of the best performing, but heavy-weight hardware solution.

This paper underscores the importance of letting application behavior guide the design of underlying support mechanisms to obtain low-complexity high-performance solutions. For example, recognizing that fast inter-core communication is not critical to efficient streaming performance, the low-cost communication support mentioned above multiplexes on the already present on-chip network for memory traffic. This allows available bandwidth to be shared between application memory requests and inter-thread operand traffic, enabling this design to efficiently support various models of application parallelism.

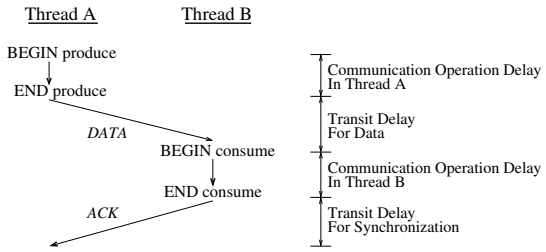

Figure 1: Transit and COMM-OP delays.

The remainder of the paper is organized as follows. Section 2 provides a characterization of high-frequency streaming applications and highlights how these applications react to the two different latency effects of inter-thread communication. Section 3 describes the design space explored and evaluated in this paper. Section 4 evaluates the performance of these design points and analyzes the results to identify the key attributes which lead to the observed performance. Section 5 shows how to improve high-frequency streaming performance without relying on dedicated interconnects or storage. Finally, the paper concludes in Section 6.

2. HIGH-FREQUENCY STREAMING

This section first provides more precise definitions for *transit delay* and *communication operation (COMM-OP) delay*. Then, it characterizes streaming programs and illustrates why transit delays are tolerated and why reducing COMM-OP delays can increase application performance. It extends Taylor, et al.'s treatment [25] of communication latency components with a discussion of their respective impact on streaming applications.

Transit delay refers to the amount of time necessary to communicate a data value from one processor core to another. This delay is *exclusive* of all the time necessary to produce a value or to initiate communication, but rather measures the effects of signal propagation delay, bus contention, network routing latency, and the like. This delay will tend to increase with the physical separation between cores or as wire delay increases.

COMM-OP delay, on the other hand, is a measure of the overhead experienced by a *single* core due to communication. More formally, the COMM-OP delay for a particular thread is the difference between the execution time of the thread with communication operations and the execution time of the same thread when communication operations have 0-latency and consume no resources.

For shared memory communication, for example, COMM-OP delay is caused by the execution of additional instructions necessary for communicating values *and* synchronizing threads. Depending on the code containing the communication and the implementation details of the memory subsystem, these extra instructions can slow down a thread by occupying valuable processor resources such as fetch bandwidth, functional units, and memory ports, and by causing execution stalls due to memory fences and interconnect contention.

Figure 1 illustrates how COMM-OP delay and transit delay affect the execution of a pair of threads communicating via a single shared buffer (*e.g.* a shared-memory variable). To send a value from thread A to thread B, thread A executes a code sequence which ensures that the shared buffer is empty, then fills the shared buffer with the value to be communicated. The time during which this happens is labeled the COMM-OP delay for thread A. Thread B will observe the value after the transit delay has elapsed. Thread B executes a code sequence that ensures the shared buffer is full,

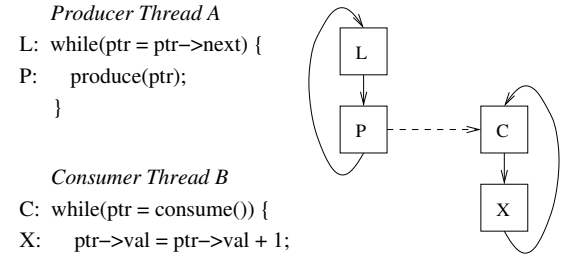

Figure 2: A pipelined streaming example.

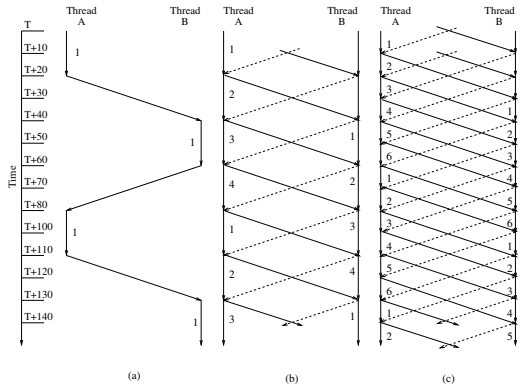

Figure 3: Effect of transit and COMM-OP delays on streaming codes.

reads the value from the buffer, and finally marks the buffer empty. Another value can be transferred using the same shared buffer *only* after the consumption notification from thread B reaches thread A (shown as an acknowledge message in Figure 1).

Streaming codes execute as concurrent, long-running, communicating threads. Figures 2 shows a two-thread streaming pipeline and its control-flow graph along with its inter-thread dependence. A key aspect of such codes is that the dependences between threads are acyclic; the threads form a pipeline with data flowing only in the forward direction.

Figure 3a illustrates the execution of the program from Figure 2 using a single shared buffer (assuming that the "L" and "X" operations take zero time). Despite the streaming nature of the application, in the naïve implementation, with only one inter-thread buffer location, the COMM-OP delay for threads A and B plus two times the full transit delay must be borne for every single stream value communicated. Notice how the application is able to complete only two iterations in the 150-cycle snapshot shown.

If, instead of a single buffer, a queue of buffers is used for communication, the threads can execute more efficiently. This is illustrated in Figure 3b. In the figure, the number adjacent to a produce (consume) operation identifies the inter-thread buffer location being written to (read from). With a queue of buffers, the COMM-OP delay of each thread is overlapped *and* useful work can be done during transit delays. The acknowledgments going from thread B to thread A are shown with dashed arrows to indicate that they no longer are on Thread A's critical path. Other than the initial time taken for the first value to arrive in thread B, transit delays do not effect the timing of the system. Compared to the non-pipelined situation with only a single buffer location, the throughput (measured as iterations per time unit) has increased by a factor of 3.5. In Figure 3b, 7 iterations are executed in 150 cycles.

```
void produce(int value) {
  // spin until tail empty
  while(q[tail].full);
  // q[tail].full == 0
  q[tail].data = value;
  q[tail].full = 1;
  tail = (tail+1)%q_size;

}
```

```
int consume() {
  // spin until head full
  while(!q[head].full);
  // q[head].full == 1
  value = q[head].data;
  q[head].full = 0;
  head = (head+1)%q_size;
  return value;
}
```

Figure 4: Produce and consume code sequences for shared-memory based software queues.

While pipelining can remove transit delays from the critical path by overlapping it with useful computation, COMM-OP delay remains critical. As Figure 3 shows, the time taken to complete one loop iteration is determined by the sum of computation time (zero in the figure) and COMM-OP delay. Reducing COMM-OP delay, therefore, can be a key enabler for improving the performance of streaming codes, even more for high-frequency streaming codes. Figure 3c shows that 14 iterations can be completed in 150 cycles by reducing the COMM-OP delay from 20 to 10 cycles. Notice, however, that to maintain peak throughput, 6 inter-thread buffer locations are necessary, rather than the 4 that were previously necessary.

Recognizing the distinction between COMM-OP delay and transit delay will serve as a guide when exploring the design space of communication mechanisms. While not shown in the examples in this section, certain designs can couple the transit delay of a particular CMP to the COMM-OP delay experienced by streaming threads. As will be discussed in Section 3, these designs should be avoided to ensure the best performance of streaming codes.

3. DESIGN SPACE

In designing high-frequency streaming support for future CMPs, architects will have to make trade-offs between hardware (area) costs, design effort, and OS costs to come up with the best design to meet desired performance goals. Any streaming support mechanism has four essential ingredients: *communication operation sequences* to specify architectural reads and writes to inter-thread queues, a *synchronization* mechanism to prevent read (write) operations on empty (full) queues, intermediate storage for queue data (*queue backing store*) before they are consumed, and an *interconnect* fabric connecting processors and various backing stores. For exposition purposes, we shall split interconnects into two sub-axes *dedicated interconnects* and *pipelined interconnects*. Although the design choice for each of these axes is orthogonal, certain design possibilities fit together more naturally than others.

3.1 Communication Operation Sequences

To avoid oversubscribing a processor core's fetch and execution resources, the communication operation sequences cannot be too long. Additionally, to avoid extending the loop critical path, the dependence height of the sequence must also remain relatively short. Finally, to enable decoupling between producer and consumer loops, the code sequences must allow queuing behavior; sequences that use only a single buffer location for communication should be avoided.

3.1.1 Software Queues Using Shared Memory

Producer/consumer communication and synchronization can be implemented on conventional shared memory multiprocessors using software queues. Code for such an implementation is shown in Figure 4. Since only a single thread will produce data into each queue and only a single thread will consume data from each queue, the head and tail pointers can be stored locally on the consumer and producer cores respectively. Additionally, no mutexes are required to protect the queue (although, the appropriate memory fence instructions are required to enforce the correct ordering of operations). Use of fine-grained condition variables allow an efficient implementation of software queues [23]. The key advantage of this methodology is that it requires no modifications to existing instruction set architectures (ISA) or microarchitectures. Its main drawback is that the code sequences to produce and consume a single datum are quite lengthy. The C code shown in Figure 4 will likely expand into many instructions. The COMM-OP delay overhead resulting from these additional instructions and dependences may offset any gains obtained by partitioning the original code among multiple threads. Further, the presence of uncounted loops in these code sequences make static ILP techniques inapplicable, and the presence of memory fence operations limit dynamic ILP and leave very little scope for performance improvements.

3.1.2 Produce and Consume Instructions

Producer/consumer communication can also be implemented by augmenting an existing ISA with special `produce` and `consume` instructions [5, 18]. The hardware is responsible for delivering values between cores and for blocking the pipeline when attempting to either write to full queues or read from empty queues. The specific hardware used to implement this is independent of the ISA as long as the queue semantics are guaranteed. This methodology overcomes many of the shortcomings experienced by software queue implementations. The produce and consume instruction sequences are reduced from tens of instructions down to a single instruction, resulting in smaller COMM-OP delays. The principal shortcoming of this methodology is the need to augment the ISA and the core microarchitecture. However, a concise expression of produce and consume semantics may be well worth the incremental core design and verification costs.

3.1.3 Register-Mapped Queues

The instruction and dependence height overhead of produce and consume operations can be further reduced using register-mapped queues [7], similar to what is used in the Raw microprocessor [24]. Rather than modifying the ISA by adding produce and consume operations, a certain portion of the register address space is reserved to refer to inter-core queues rather than traditional registers. The microarchitecture is free to implement the underlying operand network [25] using any mechanism. The main benefit is that, since any instruction can deposit its result into a communication queue and any instruction can read an operand from the communication queues, loops will contain fewer instructions and have lower dependence height than the corresponding loops with produce and consume operations. This reduced instruction count and dependence height may prove critical in resource-bound loops. On the flip side, this methodology shares its shortcomings with the explicit produce and consume instruction methodology described earlier. It additionally creates increased architectural register pressure since the register address space needs to be split between architectural registers and register-mapped queues. Consequently, for loops with a large number of live values, decreased performance due to additional spill and fill code may outweigh the advantages of eliminating produce and consume instructions.

3.2 Dedicated Interconnects

Good interconnect design is key to streaming performance. For operations consuming data or synchronization information over the interconnect, any time spent stalled due to interconnect contention

adds directly to the COMM-OP delay for that operation. Similarly, for operations producing data or synchronization information, interconnect contention may cause operations to backup in the processor pipeline, adding to the COMM-OP delay of those operations. Ideally, high-frequency streaming support should not require new routing resources, but rather, should be efficiently multiplexed with other requests on existing interconnects. However, depending on the application being run, high contention for the shared interconnect may cause communication operations to stall more often (increasing COMM-OP delays) than on a dedicated interconnect.

3.3 Pipelined Interconnects

While the transit delay of the interconnect is not important, the rate at which it can accept new requests directly affects COMM-OP delay. Pipelined interconnects increase the rate at which new requests can be serviced by the interconnect. For a non-pipelined interconnect with an N cycle latency, only one request can be carried by the interconnect every N cycles. However, an M-stage pipelined interconnect can initiate a new request every $\frac{N}{M}$ cycles. This increased throughput reduces contention for the interconnect reducing COMM-OP delay (and also improving the performance of other operations sharing the interconnect). This disparity between pipelined and non-pipelined interconnect will become more pronounced as we move to larger scale CMPs. Of course, this improved performance does not come for free. Pipelined interconnects are more complex to build than non-pipelined interconnects. Furthermore, memory systems using pipelined interconnects must use coherence protocols more sophisticated than simple snoop-based protocols to deal with multiple inflight requests in the interconnect.

3.4 Synchronization

Concurrently executing threads require a synchronization mechanism to determine when it is permissible to read from or write to a queue entry. The time from when a produce (consume) operation begins execution to when it can actually write (read) data to (from) the queue entry is called *synchronization delay*. Since every communication operation has to synchronize before reading or writing data, this delay directly affects the COMM-OP delay. The key to reducing synchronization delay for a communication operation is to ensure that the necessary synchronization information is delivered to its processor core well ahead of the operation's execution. Recall from Section 2 that pipelining communication (i.e. communicating using a queue of buffer locations rather than a single buffer location) increases the time between successive synchronizations on a single buffer location. This pipelining offers synchronization mechanisms the necessary slack to deliver synchronization information before it is read by a synchronization operation. The synchronization design options, discussed in this section, vary in the amounts of software and hardware logic, the backing store used for synchronization data and the level of OS support.

3.4.1 Software Techniques

A detailed description of software synchronization was given in Section 3.1.1. The synchronization data consists of an array of full-empty (FE) *condition variables* that are set and reset by produce and consume operations respectively. Since both produce and consume operations modify the same memory locations, with traditional caching mechanisms, synchronization delays will be significantly increased due to frequent cache misses; the first access to a particular queue slot will incur a compulsory miss and subsequent operations will incur coherence misses. (The producer and consumer may benefit from spatial locality if multiple queue entries are located in a single cache line.) To avoid such penalties, the condition variables for a queue slot should be moved from the core that writes it to the core that reads it well ahead of the read operation. Prefetching and other microarchitectural optimizations (described below) may be able to mitigate these shortcomings.

Other software queue implementations that track global queue occupancy rather than individual slot occupancy are possible. However, such mechanisms require a coarse-grain lock that guards access to the entire queue data structure. Consequently, produce and consume operations cannot occur simultaneously even if they are accessing separate portions of the queue. Such implementations will incur costly synchronization delays, since significant contention for the queue lock will exist and cache lines that store the queue occupancy must ping-pong between the producing and consuming cores.

3.4.2 Hardware Techniques

Just as in software techniques, the key to low synchronization delay is to ensure that synchronization data is maintained as close as possible to the processor core in which it is *read*. For example, maintaining full-empty (FE) bits close to the consumer core may reduce consume synchronization delay, but will increase produce synchronization delay by forcing produce operations to go all the way to the consumer core to read FE bits. Keeping the bits in a centralized location will affect both produce and consume synchronization delays. Instead, if the FE bits are replicated and one copy is maintained at the producer and consumer cores, we can achieve low produce *and* consume synchronization delays. In such a replicated setup, the two copies may be out of step with each other due to delays in propagation of updates from one core to another. However, such delays will not affect correctness (since the out-of-date information is conservative), nor will it affect performance provided there are enough empty (full) queue slots to write (read) data to (from).

Quite a few implementations for such mechanisms exist, and they are often influenced by the choice of queue backing store. For example, StreamLine [2] uses distributed occupancy counters to track memory accesses to special stream pages. The Synchronization Array [21] maintains distributed head and tail pointers to a dedicated circular buffer. By using dedicated synchronization storage specialized for streaming communication, these techniques avoid problems that stem from using the generic memory subsystem. However, in these schemes, the additional hardware synchronization state has to be added to the OS context and needs to be saved and restored on context switches. Additionally, special ISA and microarchitectural extensions and/or OS support may be needed to identify queue read/write operations to the synchronization hardware.

3.5 Queue Backing Store

The time from when a consume operation requests data from the backing store to when it receives the data contributes directly to COMM-OP delay. Just as with synchronization, this delay can be minimized by ensuring that data is stored as close as possible to the processor core that will consume the data and by ensuring that the backing store is not oversubscribed. Conversely, adding new, dedicated backing stores to a CMP design increases both the amount of die area dedicated to streaming communication and the amount of OS support required for context switches and virtualization. This section will discuss various design choices for the queue backing store in the context of this trade-off. Note, while this section discusses design options for queue data storage, the issues and mechanisms described here apply equally to synchronization storage.

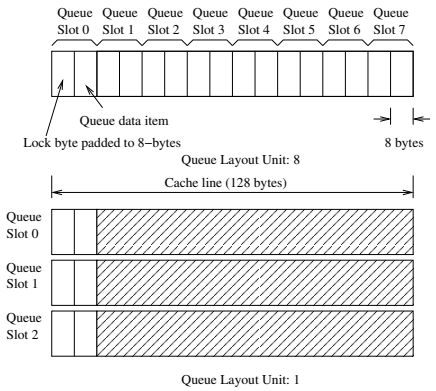

Figure 5: Two queue layouts for memory backing stores.

3.5.1 Shared Memory Store

The memory subsystem serves as a natural store for data being communicated between threads. Architectures and operating systems already provide mechanisms to share data through memory, and most memory systems are equipped with caches to buffer data close to a processor core reducing access time. Streaming communication, however, does not exhibit the same locality as traditional memory accesses. Consequently, designers must decide how streaming accesses should interact with the traditional memory hierarchy. Since streaming accesses exhibit poor temporal locality (producing and consuming threads stride across the queue, rather than access the same element multiple times), certain caches in the hierarchy may want to avoid caching streaming data since caching streaming accesses may lead to eviction of useful data.

While there is poor temporal locality within a core, writes to queue locations are soon followed by reads to the same location by other cores. Consequently, caching lines for queue storage in private caches increases coherence traffic between cores producing and consuming values. Worse still, the delay introduced by these coherence requests contribute directly to COMM-OP delay since the request is demand initiated by the consume operation. Unfortunately, avoiding caching in private caches requires that every produce or consume operation access a shared level of the memory hierarchy creating contention for its ports. Since access times to centralized caches are already typically quite large, forcing all produce and consume operations to contend for few ports will likely lead to large COMM-OP delay in addition to affecting normal memory accesses.

Streaming accesses do, however, exhibit spatial locality. Streaming produce and consume operations stride across the queue data. Consequently, caching queue storage in a core's private cache can reduce COMM-OP delay. For a consumer thread, the first access to a line will incur a large access delay, but successive consumes will be cache hits. Furthermore, if there is sufficient decoupling between a producer thread and consumer thread (*i.e.* they are writing to and reading from distinct cache lines), then coherence traffic occurs at the cache line granularity rather than for each produce and consume operation.

Unfortunately, in situations with little or no decoupling, false cache-line sharing occurs since the producer and consumer threads will be accessing nearby queue entries that fall in a single cache-line. This false sharing can create significant coherence traffic and significantly increase COMM-OP delay. Different cache-line layouts can mitigate this problem at the expense of wasted space in the caches. Two possible layouts are shown in Figure 5. In the figure, synchronization and queue data are co-located to improve locality of accesses. The layout at the top of the figure places 8 queue entries on a single cache line and can suffer from false sharing. The queue layout on the bottom of the figure, conversely, pads the size of each queue entry so that there is only entry per cache line. By construction, this layout will not experience any false sharing, but wastes large portions of the cache. The layout can be made as dense or sparse as desired. We refer to the number of queue entries per cache line as the *queue layout unit* (QLU).

Prefetching and Write-forwarding. Typically, consume operations (at least ones accessing the first queue entry on a cache line) miss in the local cache and have to fetch data from a remote cache. Such remote data fetches increase the consume COMM-OP delay compared to a local cache hit. In order to bring down this latency, two mechanisms have been proposed in the literature - remote prefetching and write-forwarding. In remote prefetching, the consumer thread issues prefetch instructions before it actually needs the data and tries to overlap the remote data fetch latency with other useful work. The consumer, however, must determine when to issue the prefetch, as overly eager prefetchers may prematurely steal cache lines from the producer's cache. This may cause the producer to slow down appreciably. The second technique, write-forwarding [14, 17, 20, 1], addresses this problem by making the producer thread forward shared cache lines to the consumer's cache *after* it is done producing its data. This way, the timing of inter-core data transfer can be optimized so that neither thread suffers any unnecessary slowdown. Write-forwarding could either be implemented with special completers on store instructions or in processors (e.g. MIPS) that support software-installable TLBs, the OS could mark certain pages as "streaming" and the memory subsystem could be modified to appropriately deal with accesses to such pages. Other mechanisms have been proposed to eagerly transfer lines from a producer's cache to a consumer's cache [19], however, the short time spans between lock access and data access present in high-frequency streaming codes makes them inappropriate for this domain.

Here, we propose a streaming-specific optimization to standard write-forwarding schemes. Stream instructions exhibit strong locality when accessing a cache line since accesses are made to consecutive stream locations. The optimization ensures that this spatial locality is not damaged by write-forwarding. Rather than forwarding the cache-line after each queue entry is filled, the cache controller forwards a line after N queue entries on the line are filled. Typically, the parameter N is set equal to the QLU so that a line is forwarded only after all queue entries on the line are filled. The implementation cost within the cache controller is minimal since it need only be parameterized with the value N and the size of each queue entry. Since accesses to successive queue entries will occur in order, accesses to certain regions of each line will initiate write-forwarding. This optimization will help reduce the average COMM-OP delay by ensuring the maximum number of cache hits possible to a line before forwarding it.

3.5.2 Dedicated Store

A backing store implemented with dedicated hardware is another possibility. Having a dedicated backing store is advantageous as it does not pollute the memory subsystem with short-lived streaming data. It also helps avoid all streaming related coherence traffic. By ensuring that streaming traffic does not contend with shared memory requests, dedicated stores can prevent normal memory traffic from increasing COMM-OP delays (or streaming operations from decreasing the performance of normal memory operations).

While dedicated stores can potentially improve performance, they

do not come without a cost. Dedicated stores consume valuable die area possibly reducing available on-chip cache memory and negatively impacting the performance of non-streaming sections of applications. Additionally, the contents of the dedicated store become part of the OS context for a process. As such, OS and hardware support is necessary to context switch and virtualize these resources.

Centralized Dedicated Store. A centralized dedicated store adds a single streaming-specific memory to the CMP. With this design all cores can share all the storage added to the CMP. Unfortunately, this design suffers from scalability problems as more cores try to access the single structure. Additionally, since the structure is centrally located, for all but the smallest CMPs, the structure will be farther from cores than the local caches. Consequently, the time required to access the store will likely be larger than a local cache hit. This translates to a comparatively larger COMM-OP delay.

Distributed Dedicated Store. Alternatively, streaming-specific memories can be added to each core in the CMP, rather than adding a single central structure. This design scales better than the centralized dedicated store since a single common structure does not need to be accessed by all cores. Additionally, each piece of the distributed store can be located close to the consuming processor reducing the COMM-OP delay for consume operations (recall that a distant backing store does not increase the COMM-OP delay for produce operations). Unfortunately, this design prevents cores from sharing the added storage. Consequently, more die area may be consumed for the dedicated store.

3.5.3 Network Backed Queues

Intermediate nodes in an on-chip network, often buffer data to implement pipelined interconnects. By preserving data ordering they act as effective FIFOs [25]. They inherit all the advantages of dedicated stores. Their distributed nature also makes them scalable. Unfortunately, the amount of decoupling available to the executing threads is directly proportional to the physical separation of their cores on the chip. The larger the separation the more the available decoupling. Relying solely on this storage can affect performance as threads executing on nearby cores will not get sufficient decoupling to tolerate variable latency stalls in the individual threads. Additionally, when network buffers are the sole carriers of inter-thread architectural state, the OS overhead for context switches becomes more pronounced. While switching out a consumer thread, the OS has to check network buffers along the paths from every producer to this consumer before doing the switch. Alternatively, every time data arrives at a node for a thread that has been switched out, an interrupt could be triggered to make the OS append the incoming data to the swapped out context state.

4. EVALUATION

In this section, we select four important points from the design space described in the previous section. These points represent design variants ranging from existing commercial processor designs to designs leveraging heavy-weight dedicated streaming hardware to maximize the performance of streaming codes. All selected design points ensure backward compatibility with legacy software. Using these design points, we empirically illustrate that streaming codes do, in fact, tolerate transit delay. We then evaluate how much performance can be improved over existing commercially available systems by streaming-aware designs. The results and analysis from this section motivate optimizations described and analyzed in the next section.

4.1 Systems Studied

The four design points explored were:

EXISTING. This design point is representative of existing commercial CMPs. This design will serve as our baseline for measuring the hardware cost and operating system impact of other designs.

MEMOPTI. This variant will illustrate the efficacy of write-forwarding, a low-impact memory subsystem optimization. This design requires little additional hardware (cache modifications to support write-forwarding). We do not write-forward to L1 caches to avoid polluting it with short-lived streaming data.

SYNCOPTI. This variant will illustrate the benefits of optimizing communication operation sequences and synchronization, while still relying on the memory subsystem for queue backing stores and core-to-core interconnect. Since this design point does not resemble any design previously proposed in the literature, it will be described in more detail in Section 4.2. The design requires modifying core pipelines to execute `produce` and `consume` instructions, write forwarding logic (with the locality enhancements described in Section 3.5.1) and synchronization counters in the processor caches, and OS support to context switch the synchronization counters. Here too, we avoid write-forwarding to L1.

While this design does introduce more hardware than the previous design, it remains fairly light weight. This design reuses the L2-L3 memory bus, a critical component of CMP architectures, rather than introducing a new network like HEAVYWT. Such an approach makes optimal use of the available on-chip transistors; the single on-chip network can be provisioned according to the total system bandwidth requirements without regard to how such traffic is generated (application memory requests or inter-thread operand requests). This generality makes the solution appealing since it has potential to support various models of application parallelism. Additionally, use of memory as a backing store avoids introducing new dedicated stores, allows flexible queue sizing, *and* greatly reduces OS context-switch and virtualization costs.

HEAVYWT. This variant represents the performance achievable by hardware-heavy mechanisms such as the FIFOs provided by the scalar operand networks in Raw [25] or the synchronization array (SA) [21] hardware. It combines the point along each design axis that should offer greatest performance without regard for hardware cost or OS impact. In addition to core modifications to execute `produce` and `consume` instructions, HEAVYWT introduces additional dedicated distributed on-chip queue backing store and a new interconnect network to connect processor cores to this backing store. The contents of the backing store and in-flight data buffered in the network must be part of a process's context. Consequently, this design variant *also* requires OS modifications to context switch this state. Since this state is concurrently updated by multiple threads belonging to the same process, the OS implications will be more far reaching than for the other design variants.

4.2 SYNCOPTI Implementation

This design attempts to create an efficient streaming-tuned message passing implementation atop a shared-memory CMP. Others have proposed mechanisms to implement efficient message passing in shared-memory multiprocessors [2, 5, 8, 11, 20]. However, message setup overhead in these designs make them in appropriate for high-frequency streaming applications.

In this technique, `produce` and `consume` instructions are dynamically renamed to unique memory addresses. Microarchitectural stream address logic tracks accesses to all queues and assigns consecutive stream addresses to all accesses to the same queue number, modulo the queue size. The latency of this logic is entirely

overlapped with the L1 cache access latency. Per-queue hardware occupancy counters maintained at the L2 controller provide synchronization. A producer (consumer) core updates its occupancy counters after successfully executing a `produce` (`consume`) instruction or after snooping occupancy updates from the bus. A `produce` (`consume`) instruction is allowed to access the L2 cache if and only if the occupancy counter corresponding to its stream does not indicate a full (empty) queue. Write-forward messages are used by the consumer core to update its occupancy counters. When the last queue item ("last" depends on the queue layout) from a given streaming line is read by a `consume` instruction, the consumer core sends out a message on the bus to inform the producer's occupancy-tracker of how many `consume` instructions were serviced from that particular line. When a streaming cache line is evicted from an L2 cache, then the cache once again puts out on the bus the number of queue items produced into (or consumed from) the line for its counterpart to update its occupancy counters. When the producer thread wraps around, it is stalled until all queue items from the corresponding line have been consumed by the consumer, to avoid damaging spatial locality in the consumer. Finally, since no write-forward messages will be sent when a stream terminates midway through a cache line, `consume` requests initiate an L3 access after a time-out to elicit a writeback from the producer core, to obtain the remaining queue items and avoid deadlock. Although the proposed implementation is a bus-based one, through simple modifications to the occupancy update protocol, this can be adapted to network-based interconnects of future CMPs.

In order to cope with increased inter-thread streaming traffic due to multiple threads or more complex communication patterns (for example, scatter-gather), the memory network arbiter can be modified to favor application memory requests over inter-thread operand traffic (a simple way to do this is to just look at the memory area being accessed). While application memory performance remains unaffected, pipelined inter-thread communication helps tolerate delays due to increased contention.

4.3 Experimental Setup

Despite the difficulty of hand-partitioning applications and the fact that automatic thread-pipelining is a relatively nascent research area, we were able to find a good number of applications for our experiments. The first set consisted of applications from SPEC-CPU2000, Mediabench [13] and the Unix utility 'wc'. The hottest loop in each of these applications (given in Table 1) were automatically parallelized with the DSWP algorithm [15] by a modified version of OpenIMPACT [9]. All compilations were targeted for a dual-core Itanium 2 CMP and hence the applications executed as two threads. The second set consisted of two StreamIt [22] benchmarks (with C source) that were each hand-parallelized into two threads to mirror the corresponding StreamIt programs.

We use a shared-bus dual-core Itanium 2 CMP as the baseline for our evaluation. Note that, even though the techniques have been evaluated on a dual-core CMP, the pairwise nature of inter-thread interactions in pipelined streaming codes means that the insight and conclusions drawn from a dual-core setup are just as valid for larger-scale CMPs of the future. Our simulation infrastructure was built on top of a validated cycle-accurate Itanium 2 processor [10] performance model (IPC accurate to within 6% of real hardware for the benchmarks measured [16]) using the Liberty Simulation Environment [27]. Note that all comparisons are *only* for the multi-threaded loops. Table 2 provides details about the baseline simulation model.

All designs used 64 queues of depth 32 unless otherwise mentioned (not all queues were used by each application). For all designs using shared memory backing stores, the queue layout unit was 8. Experiments were also conducted with QLU 1, but since performance was uniformly better with QLU 8 (even with software-based stride prefetching), the results have been omitted for brevity. For all designs using write-forwarding, lines were forwarded only to the cores' private L2 caches and forwarding was initiated only after all queue entries on the line had been written. For configurations with a dedicated backing store, the backing store was located in the consumer core, but queue synchronization counters were maintained at both the producer and consumer core. The dedicated store could service 4 concurrent operations per cycle and was connected to remote cores by a dedicated pipelined interconnect. Within a consuming core, the consume-to-use latency was 1-cycle. If not otherwise mentioned, the interconnect latency was 1-cycle.

Benchmark	Function	% Exec. Time
`wc`	`cnt`	100%
`adpcmdec`	`adpcm_decoder`	98%
`183.equake`	`smvp`	68%
`181.mcf`	`refresh_potential`	30%
`epicdec`	`read_and_huffman_decode`	21%
`179.art`	`match`	20%
`256.bzip2`	`getAndMoveToFrontDecode`	17%

Table 1: Benchmark Loop Information.

Core	Functional Units - 6-issue, 6 ALU, 4 Memory, 2 FP, 3 Branch L1I Cache - 1 cycle, 16 KB, 4-way, 64B lines L1D Cache - 1 cycle, 16 KB, 4-way, 64B lines, Write-through L2 Cache - 5,7,9 cycles, 256KB, 8-way, 128B lines, Write-back Maximum Outstanding Loads - 16
Shared L3 Cache	$>$ 12 cycles, 1.5 MB, 12-way, 128B lines, Write-back
Main Memory latency	141 cycles
Coherence	Snoop-based, write-invalidate protocol
L3 Bus	16-byte, 1-cycle, 3-stage pipelined, split-transaction bus with round robin arbitration

Table 2: Baseline Simulator.

The code sequences for the load-store based produce and consume operations have been highly tuned to contain the minimal number of instructions possible. Despite that, the software overhead for a communication operation was 10 instructions (6, 1 and 3 instructions for synchronization, data transfer and stream address update respectively) with a dependence height of 4. The overhead for the `produce-consume` instructions based versions was just the one instruction for data transfer. On an in-order machine such as the Itanium 2, we see that these overheads tend to contribute significantly towards the overall execution time, especially for really tight loops.

4.4 Results and Analysis

To understand the decoupling present in our applications and to see the effect of transit delay on pipelined streaming communication, we present in Figure 6 a normalized execution time comparison of three HEAVYWT variants. They differ only in the end-to-end latency of their dedicated streaming interconnects. The leftmost bar corresponds to a 1-cycle end-to-end latency (default HEAVYWT) and the middle bar to a 10-cycle latency. The rightmost bar corresponds to a 10-cycle interconnect latency with a 64-entry queue. All other design parameters are held constant. Overall, we do not see much of a difference between the first two bars.

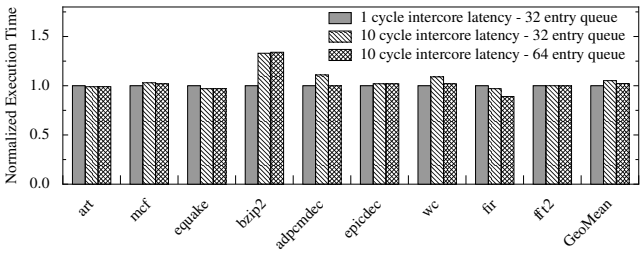

Figure 6: Effect of transit delay on streaming codes.

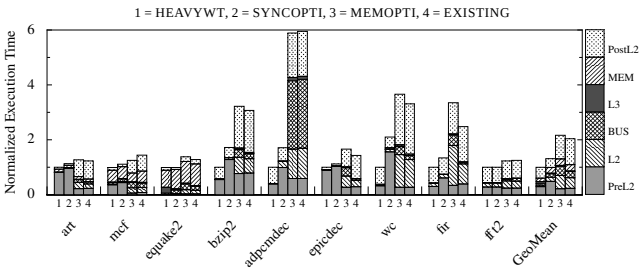

Figure 7: Normalized execution times for each design point.

However, for `256.bzip2` we see a significant slowdown (33%). This is because the loop that is parallelized in this benchmark is actually a two deep loop-nest with both the inner and outer loops requiring inter-thread communication. The reason for the slowdown is because the outer loop's `consume` instructions (in the consumer thread) tend not to get serviced quickly as the producer thread can get to the corresponding outer loop `produce` instructions only after it is done with all of its inner loop iterations. So, due to poor decoupling at the outer loop level, the data transfers could not be pipelined, leading to the slowdown. In `179.art`, `183.equake`, and `fir` the 10-cycle latency turns out to be better because the longer latency in a pipelined interconnect in effect becomes extra storage on the network and this way the producer core does not stall on queue full conditions as frequently. The slight slowdown in the other benchmarks is caused by the extra delay for a synchronization acknowledgment to go from the consumer core to the producer core. This can be improved by making the queue size bigger as can be seen from the rightmost bars.

Figure 7 shows the normalized execution times for the producer threads of all benchmarks. From left to right, for each benchmark, the bars correspond to HEAVYWT, SYNCOPTI, EXISTING and MEMOPTI respectively. Due to space constraints, we omit the graphs for the consumer core. The overall performance

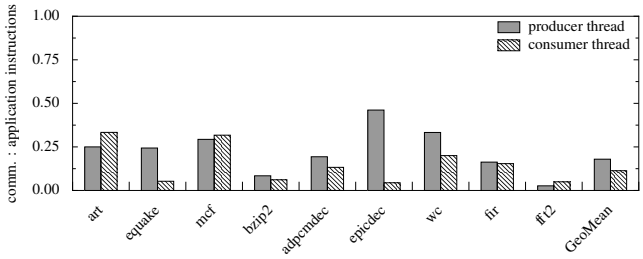

Figure 8: Ratio of the # of dynamic communication instructions to application instructions for producer and consumer threads.

of the consumer core was the same as for the producer, except that its component breakdowns differed due to different computation and communication instructions. The component-wise breakdowns represent non-overlappable stalls that contributed directly to the critical path delay in the respective sections of the machine. Since we are dealing primarily with the memory subsystem, we aggregate the delays for stages preceding (main pipe) and following (L1 fill and writeback) the L2 cache into *PreL2* and *PostL2* respectively. *L2*, *L3* and *MEM* represent the time spent in the L2, the L3 and the main memory respectively, *BUS* is the total time spent on the shared bus (including arbitration, snoops, requests, and data transfers).

The figures show that HEAVYWT and SYNCOPTI perform better than MEMOPTI and EXISTING for all benchmarks. It is obvious from design why HEAVYWT is the best overall (since it provides the lowest COMM-OP delay). SYNCOPTI closely trails HEAVYWT across all the benchmarks. This is expected since SYNCOPTI and HEAVYWT are identical is all respects, except in their queue backing stores. However, there is still a small difference between the SYNCOPTI and HEAVYWT bars across all benchmarks and in fact the difference is pretty significant in `wc`. After a careful examination of the pipeline behavior of these benchmarks, we identified the main reason for the slowdown. The average `consume`-to-use latency in SYNCOPTI is at least 6 cycles (since synchronization happens in L2 following a 2-cycle stream address generation), whereas it is 1 cycle in HEAVYWT. The higher COMM-OP delay results in the consumer performing slower in SYNCOPTI than in HEAVYWT. This in turn delays freeing up of queue slots, thereby ultimately slowing down the producer thread. For `wc`, the reason why SYNCOPTI is almost twice as slow as HEAVYWT is because the streaming loop is very tight. With three consume operations per loop iteration, the overhead turns out to be a significant factor.

While HEAVYWT incurs no memory system overhead (by design), SYNCOPTI does equally well too, as can be seen by the *L2* and *BUS* components. Since synchronization counters are efficiently maintained and updated in a distributed fashion, SYNCOPTI avoids unnecessary cache line ping-ponging between cores. The only extra memory traffic stems from uni-directional queue line transfers and bulk ACK notifications for counter updates. However, since MEMOPTI and EXISTING have to explicitly modify condition variables and communicate them in both directions, their memory system performances are significantly poorer. Since SYNCOPTI, MEMOPTI and EXISTING all effect data transfers through the memory subsystem, one might expect their breakdowns to be somewhat similar. However, that is not the case, because, in MEMOPTI and EXISTING, instructions recirculate through the OzQ[1] when they cannot issue because of port contention or to respect memory fence semantics. Further, when a produce operation tries to produce into a full queue, the spin lock instructions keep flowing through the pipeline till the produce happens. Whereas, in SYNCOPTI, a `produce` instruction takes up one OzQ slot and remains dormant till it goes past the synchronization phase in its state machine. Often, this causes the OzQ to fill up leading to backpressure in the pipeline, resulting in a larger *preL2* component. Finally, the greater intrinsic schedule height for software queues, causes MEMOPTI and EXISTING to have larger *postL2* components than SYNCOPTI since fewer instructions execute and writeback in SYNCOPTI. Hence the differences in the breakdowns.

For a number of benchmarks, EXISTING performs better than

[1]An ordered queue of outstanding transactions, in the Itanium 2's L2 controller, whose entries also serve as miss status holding registers (MSHRs).

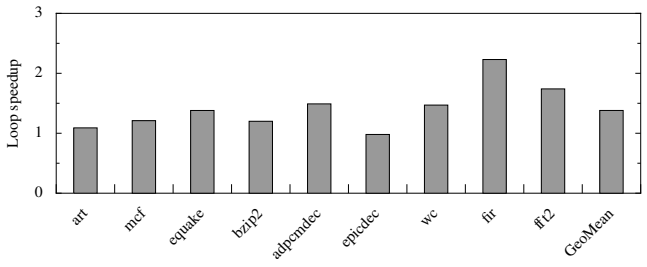

Figure 9: Speedup of optimized loops in HEAVYWT over single-threaded execution.

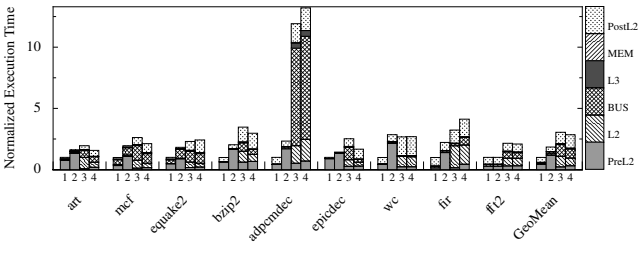

Figure 10: Effect of increased transit delay on pipelined and unpipelined interconnects (transit delay = 4 cycles).

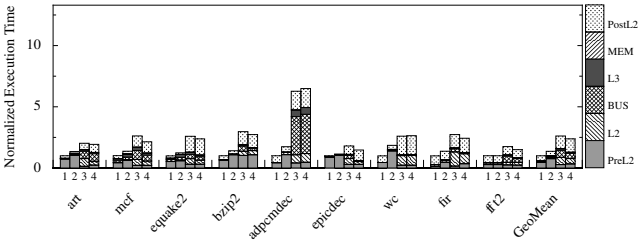

Figure 11: Effect of increased increased interconnect bandwidth (transit delay = 4 cycles, bus width = 128 bytes).

MEMOPTI. When the number of write-forwarding instructions (communication operations) is high, the OzQ in MEMOPTI fills up with write-forwarding requests causing all regular requests to run out of L2 ports as the earlier instructions actively recirculate through the L2 and occupy ports (bigger *L2* components for MEMOPTI in Figure 7). Further, the presence of a memory fence instruction per communication operation enforces in-orderness among recirculating memory requests. In EXISTING, when a remote core needs queue data, it elicits a writeback from the cache that owns the line. Since the local L2 controller accords higher priority to external coherence requests (like writeback requests) over local recirculates, the writeback requests are able to get cache ports and complete the transfers quicker compared to write-forwarding in MEMOPTI. This leads to lower average COMM-OP delays leading to improved performance. Note that this phenomenon is unique to this particular implementation of the L2 controller. Other implementations could recirculate less aggressively, causing lower port contention leading to improved performance.

Overall, a major factor contributing to the improved performance of HEAVYWT and SYNCOPTI over MEMOPTI and EXISTING is the *postL2* component. MEMOPTI and EXISTING simply commit many more instructions due to the software overhead for synchronization and address generation and this directly causes them to perform worse than HEAVYWT and SYNCOPTI. Figure 8, which has a plot of the ratios of the dynamic counts of communication and synchronization instructions to application instructions for both the producer and consumer threads for codes with `produce-consume` instructions, shows that on the average, a communication is required once every 5 to 20 dynamic application instructions. Given this high communication frequency, the 10 instruction sequence, required per communication with software queues, proves to be a significant overhead and detrimentally affects software queue performance. The performance of SYNCOPTI is in between that of the HEAVYWT mechanism and the EXISTING and MEMOPTI mechanisms. On the average, it has 1.6x speedup over EXISTING and MEMOPTI and a modest 31% slowdown relative to HEAVYWT. Figure 9 shows the speedup of HEAVYWT over single-threaded codes for the optimized loops. Notice that the geomean speedup of HEAVYWT over single-threaded codes is 29%. This means the communication overhead for the other mechanisms actually *negates* parallelization benefits and causes multithreaded execution to perform *worse* than single-threaded execution. These basic experiments demonstrate the importance of efficient communication support for high-frequency streaming.

4.5 Sensitivity Study

In order to evaluate the sensitivity of the four techniques to increased wire delays of future CMPs, we repeated the experiments with a bus latency of 4 CPU cycles. For HEAVYWT, we increased the end-to-end latency of the dedicated interconnect to 4 cycles as well. The execution time breakdown is presented in Figure 10. Benchmarks with tight loops (`adpcmdec`, `wc` and `epicdec`) tend to be affected the most due to the increase in bus latency. However, even for relatively larger loops in `181.mcf` and `183.equake` the *BUS* component turns out to be pretty significant (due to increased arbitration delays). This is not surprising, given that it takes 8 ($\frac{linesize(128)}{buswidth(16)}$) bus cycles for a line to be transferred on the bus. With a bus latency of 4 CPU cycles, it takes 32 CPU cycles for line transfers. This causes requests to backlog leading to large arbitration delays. Further, `181.mcf` and `183.equake` are memory-intensive applications and tend to access the L3 cache frequently, making them sensitive to bus delays.

To see if interconnect bandwidth is the problem, we ran another set of experiments with a bus width of 128 bytes (equal to cache line size) holding the latency at 4 CPU cycles (peak bandwidth of 32 bytes per cycle). This change significantly eases contention leading to lower arbitration delays as seen from the *BUS* components in Figure 11. This highlights the importance of *interconnect bandwidth* for high-frequency streaming. Although building a 128-byte-wide interconnect can be expensive, the same benefits can be had by using a pipelined interconnect with equal bandwidth.

5. OPTIMIZATIONS

Section 4 shows that the performance of SYNCOPTI trails the performance of HEAVYWT due to the large `consume`-to-use latency that delayed the initiation of future `consume` instructions causing the producer thread to eventually stall. Based on these observations, we evaluated two optimizations to SYNCOPTI. First, in order to avoid frequent producer thread stalls due to queue-full conditions, we increased the queue size to 64 entries (up from 32 entries), and increased the QLU to pack 16 8-byte queue items per cache line (Q64). Second, in order to reduce the average `consume`-to-use latency, we evaluated the use of a special fully associative 1KB stream cache (SC). Improving consumer performance indi-

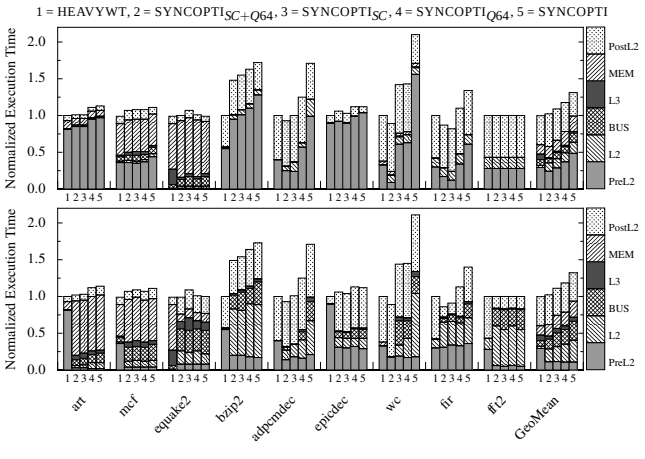

Figure 12: Effect of streaming cache and queue size on producer (above) and consumer (below).

rectly improves producer performance by avoiding frequent queue full stalls. While this cache does add additional storage to each processor core, this storage amounts to less than 1% of the storage used for the dedicated queue backing store. The proposed stream cache works as follows. When cache lines mapped to queues are forwarded from the producer's L2 to the consumer's L2, after filling the consumer's L2, the memory address is reverse mapped to a queue address (a two-tuple of queue number and queue slot) that is used to fill the streaming cache. By using a different address space, `consume` instructions are now able to access queue data, without going through TLB lookup, memory address generation, etc. Stream cache entries are invalidated by `consume` instructions that hit. If the stream cache is full, then write-forwarding fills are ignored. In this modified SYNCOPTI design, `consume` instructions continue to go to the L2, even if they are serviced by the stream cache, to ensure the synchronization counters are updated and the producer core is informed of these updates. If a `consume` instruction misses in the streaming cache, then it is handled just as it was in the original SYNCOPTI model.

Note that this optimization requires stream address generation logic in the processor pipeline (akin to HEAVYWT) to rename `consume` instructions to the correct queue addresses to index into the stream cache. However, this is still better than HEAVYWT, since SYNCOPTI shares the L3 bus, while HEAVYWT requires extra interconnects connecting the cores to the synchronization array, which can be expensive [12].

We evaluated both these optimizations in isolation and together. Figure 12 presents the breakdown for the producer (above) and consumer (below) cores. As we go from right to left, SYNCOPTI$_{Q64}$ improves the producer by reducing stalls (smaller *preL2*) and improves the consumer by providing improved cache locality (smaller *L2*) through a denser queue layout. Next, SYNCOPTI$_{SC}$ lowers `consume`-to-use latency and improves the performance of both cores. SYNCOPTI$_{SC+Q64}$ combines the benefit of both by further reducing stalls in the producer and lowering the consumer's *L2* component. It is able to achieve performance equaling HEAVYWT at times even performing better, achieving a 2x speedup over EXISTING and MEMOPTI mechanisms and bridging the gap with HEAVYWT to just 2%.

6. CONCLUSION

Pipelined streaming has emerged as a viable technique for parallelizing general-purpose programs. Its success, however, hinges on efficient underlying support for inter-thread communication, particularly for *high-frequency* streaming codes. We argued and quantitatively demonstrated that if communication is pipelined, high-frequency streaming programs can tolerate growing transit delays. Streaming codes, however, are extremely sensitive to COMM-OP delays, the recurring intra-core overhead associated with communication.

Using this insight, the paper characterized the design space of inter-core streaming communication mechanisms and quantitatively evaluated four points from the design space which offer a trade-off between implementation cost and application performance. The results showed that, SYNCOPTI enhanced with a streaming cache, a novel design proposed in this paper, achieves 98% of the speedup of a heavy-weight design (a speedup of 2.0 over existing commercial CMPs) while using only 1% of the additional on-chip storage hardware. This design exemplifies the value of optimizing communication systems to reduce COMM-OP delay while not expending design effort to reduce transit delay.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. Additionally, we thank the anonymous reviewers for their insightful comments. This work has been supported by Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of Intel Corporation.

7. REFERENCES

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, pages 204–215, February 1997.

[2] G. T. Byrd. *Communication Mechanisms in Shared Memory Multiprocessors*. PhD thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, 1998.

[3] E. Caspi, A. DeHon, and J. Wawrzynek. A streaming multi-threaded model. In *Proceedings of the Third Workshop on Media and Stream Processors*, December 2001.

[4] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–248, 2005.

[5] M. I. Frank and M. K. Vernon. A hybrid shared memory/message passing parallel machine. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages 232–236. CRC Press, August 1993.

[6] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

[7] T. Gross and D. O'Halloron. *iWarp, Anatomy of a Parallel Computing System*. MIT Press, 1998.

[8] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of message passing and shared memory in the

Stanford FLASH multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50. ACM Press, 1994.

[9] The IMPACT compiler. Web site: http://www.crhc.uiuc.edu/IMPACT, June 2004.

[10] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA, 2002.

[11] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating message-passing and shared-memory: early experience. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63. ACM Press, May 1993.

[12] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 408–419. IEEE Computer Society, June 2005.

[13] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[14] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[15] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, November 2005.

[16] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation*, June 2005.

[17] D. Poulsen. *Memory Latency Reduction via Data Prefetching and Data Forwarding in Shared-Memory Multiprocessors*. PhD thesis, University of Illinois, Urbana, IL, 1994.

[18] B. R. Preiss and V. C. Hamacher. A cache-based message passing scheme for a shared-bus multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 358–364. IEEE Computer Society Press, 1988.

[19] R. Rajwar, A. Kagi, and J. R. Goodman. Inferential queueing and speculative push for reducing critical communication latencies. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 273–284. ACM Press, June 2003.

[20] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, page 62. ACM Press, 1995.

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

[22] StreamIt benchmarks. Web site: http://cag.csail.mit.edu/streamit/shtml/benchmarks.shtml.

[23] M. Takesue. Software queue-based algorithms for pipelined synchronization on multiprocessors. In *Proceedings of the 2003 International Conference on Parallel Processing Workshops*, October 2003.

[24] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuit and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.

[25] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, February 2005.

[26] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 12th International Conference on Compiler Construction*, 2002.

[27] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 271–282, November 2002.