Comparative Evaluation of Memory Models for Chip Multiprocessors

JACOB LEVERICH, HIDEHO ARAKIDA, ALEX SOLOMATNIKOV,
AMIN FIROOZSHAHIAN, MARK HOROWITZ, and CHRISTOS KOZYRAKIS
Stanford University

There are two competing models for the on-chip memory in Chip Multiprocessor (CMP) systems: *hardware-managed coherent caches* and *software-managed streaming memory*. This paper performs a direct comparison of the two models under the same set of assumptions about technology, area, and computational capabilities. The goal is to quantify how and when they differ in terms of performance, energy consumption, bandwidth requirements, and latency tolerance for general-purpose CMPs. We demonstrate that for data-parallel applications on systems with up to 16 cores, the cache-based and streaming models perform and scale equally well. For certain applications with little data reuse, streaming scales better due to better bandwidth use and macroscopic software prefetching. However, the introduction of techniques such as hardware prefetching and nonallocating stores to the cache-based model eliminates the streaming advantage. Overall, our results indicate that there is not sufficient advantage in building streaming memory systems where all on-chip memory structures are explicitly managed. On the other hand, we show that streaming at the programming model level is particularly beneficial, even with the cache-based model, as it enhances locality and creates opportunities for bandwidth optimizations. Moreover, we observe that stream programming is actually easier with the cache-based model because the hardware guarantees correct, best-effort execution even when the programmer cannot fully regularize an application's code.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Performance, Design

Additional Key Words and Phrases: Chip multiprocessors, cache coherence, streaming memory, parallel programming, locality optimizations

ACM Reference Format:

Leverich, J., Arakida, H., Solomatnikov, A., Firoozshahian, A., Horowitz, M., and Kozyrakis, C. 2008. Comparative evaluation of memory models for chip multiprocessors. ACM Trans. Architec. Code Optim. 5, 3, Article 12 (November 2008), 30 pages. DOI = 10.1145/1455650.1455651 http://doi.acm.org/10.1145/1455650.1455651

Authors' address:

1. INTRODUCTION

The scaling limitations of uniprocessors [Agarwal et al. 2000] have led to an industry-wide turn towards chip multiprocessor (CMP) systems. CMPs are becoming ubiquitous in all computing domains. Unlike uniprocessors, which have a dominant, well-understood model for on-chip memory structures, there is no widespread agreement on the memory model for CMP designs. The choice of memory model can significantly impact efficiency in terms of performance, energy consumption, and scalability. Moreover, it is closely coupled with the choice of parallel programming model, which in turn affects ease of use. Although it is possible to map any programming model to any memory model, it is typically more efficient if the programming model builds upon the basic principles of the memory model.

Similar to larger parallel systems [Culler et al. 1999], there are two competing memory models for contemporary CMP systems: *hardware-managed, implicitly-addressed, coherent caches* and *software-managed, explicitly-addressed, local memories* (also called *streaming memory*). With the cache-based model, all on-chip storage is used for private and shared caches that are kept coherent by hardware. The advantage is that the hardware provides best-effort locality and communication management, even when the access and sharing patterns are difficult to statically analyze. With the streaming memory model, part of the on-chip storage is organized as independently addressable structures. Explicit accesses and DMA transfers are needed to move data to and from off-chip memory or between two on-chip structures. The advantage of streaming memory is that it provides software with full flexibility on locality and communication management in terms of addressing, granularity, and replacement policy. Since communication is explicit, it can also be proactive, unlike the traditionally reactive behavior of the cache-based model. Hence, streaming allows software to exploit producer-consumer locality, avoid redundant write-backs for temporary results, manage selective data replication, and perform application-specific caching and macroscopic prefetching. Streaming eliminates the communication overhead and hardware complexity of the coordination protocol needed for cache coherence. On the other hand, it introduces software complexity, since either the programmer or the compiler must explicitly manage locality and communication.

Traditional desktop and enterprise applications are difficult to analyze and favor cache-based systems. In contrast, many upcoming applications from the multimedia, graphics, physical simulation, and scientific computing domains are being targeted by both cache-based [Andrews and Backer 2005; Yeh 2005] and streaming [Ahn et al. 2004; Taylor et al. 2004; Dally et al. 2003; Gschwind et al. 2005; Machnicki 2005; Khailany et al. 2008] systems. The debate is particularly interesting vis-à-vis contemporary game consoles. The CMPs for the Xbox360 and PlayStation 3 are similar in terms of the *type* of processors they use: dual-issue, in-order RISC cores with 128-bit SIMD support [Andrews and Backer 2005; Gschwind et al. 2005]. However, they differ dramatically in their on-chip memory model, as Xbox360's Xenon processor (Waternoose) is a cache-based CMP while PlayStation 3's Cell processor is a streaming memory CMP.

Hence, it is interesting to evaluate if streaming provides specific benefits to this domain, or if using a cache-based approach across all application domains should be preferable.

The goal of this article is to compare the efficiency of the two memory models under the same set of assumptions about technology, area, and computational capabilities. Specifically, we are interested in answering the following questions: How do the two models compare in terms of overall *performance* and *energy consumption*? How does the comparison change as we *scale* the number or compute throughput of the processor cores? How sensitive is each model to *bandwidth or latency variations*? Are *multilevel memory hierarchies* equally important for both models? We believe that such a direct comparison will provide valuable information for the CMP architecture debate and generate some guidelines for the development of future systems.

The major conclusions from our comparison are:

- For data-parallel applications on systems with up to 16 cores and abundant data reuse, the two models perform and scale equally well. Caches are as effective as software-managed memories at capturing locality and reducing average memory access time. For some of these applications, streaming has an energy advantage of 10% to 25% over write-allocate caches because it avoids superfluous refills on output data streams. Using a no-write-allocate policy for output data in the cache-based system reduces the streaming advantage.
- For applications without significant data reuse, macroscopic prefetching (double-buffering) provides streaming memory systems with a performance advantage when we scale the number and computational capabilities of the cores. The use of hardware prefetching with the cache-based model eliminates the streaming advantage for some latency-bound applications. There are also cases where streaming performs worse, such as when it requires redundant copying of data or extra computation to manage local stores.
- Our results indicate that a pure streaming memory model is not sufficiently advantageous at the memory system level. With the addition of prefetching and nonallocating writes, the cache-based model provides similar performance, energy, and bandwidth behavior. On the other hand, we found that "streaming" at the programming model level is very important, even with the cache-based model. Properly blocking an application's working set, exposing producer-consumer locality, and identifying output-only data leads to significant efficiency gains. Moreover, stream programming leads to code that requires coarser-grain and lower-frequency coherence and consistency in a cache-based system. This observation will be increasingly relevant as CMPs scale to much larger numbers of cores.
- Finally, we observe that stream programming is actually easier with the cache-based model because the hardware guarantees correct, best-effort execution, even when the programmer cannot fully regularize an application's code. With the streaming memory model, the software must orchestrate locality and communication perfectly, even for irregular codes.

The remainder of this article is organized as follows: Section 2 summarizes the two memory models and discusses their advantages and drawbacks. Section 3 presents the architectural framework for our comparison and Section 4 describes the experimental methodology. Section 5 analyzes our evaluation results. Section 6 focuses on streaming at the programming level and its interactions with CMP architecture. Section 7 addresses limitations in our study. Finally, Section 8 reviews related work and Section 9 concludes the article.

2. ON-CHIP MEMORY MODELS FOR CHIP MULTIPROCESSORS

General-purpose systems are designed around a *computational model* and a *memory model*. The computational model defines the organization of execution resources and register files and may follow some combination of the the superscalar, VLIW, vector, or SIMD approaches. The memory model defines the organization of on-chip and off-chip memories as well as the communication mechanisms between the various computation and memory units. The two models are linked, and an efficient system is carefully designed along both dimensions. However, we believe that the on-chip memory model creates more interesting challenges for CMPs. First, it is the one that changes most dramatically compared to uniprocessor designs. Second, the memory model has broader implications on both software and hardware. Assuming we can move the data close to the execution units in an efficient manner, it is not difficult to select the proper computational model based on the type(s) of parallelism available in the computation kernels.

For both the cache-based and streaming models, certain aspects of the on-chip memory system are set by VLSI constraints such as wire delay [Ho et al. 2001]. Every node in the CMP system is directly associated with a limited amount of storage (first-level memory) that can be accessed within a small number of cycles. Nodes communicate by exchanging packets over an on-chip network that can range from a simple bus to a hierarchical structure. Additional memory structures (second-level memory) are also connected to the network fabric. The system scales with technology by increasing the number of nodes, the size of the network, and the capacity of second-level storage. Eventually, the scale of a CMP design may be limited by off-chip bandwidth or energy consumption [Horowitz and Dally 2004].

Although they are under the same VLSI constraints, the cache-based and streaming models differ significantly in the way they manage data locality and inter-processor communication. As shown in Table I, the cache-based model relies on hardware mechanisms for both, while the streaming model delegates management to software.

The rest of this section overviews the protocol and operation for each memory model for CMP systems. The discussion below separates the two memory models from the selection of a computation model. The streaming memory model is general and has already been used with VLIW/SIMD systems [Ahn et al. 2004], RISC cores [Taylor et al. 2004], vector processors [Lin 2004], and even DSPs [Machnicki 2005]. The cache-based model can be used with any of these computation models as well.

Table I. The Design Space for on-Chip Memory for CMPs

		Communication	
		H/W	S/W
Locality	H/W	**Coherent Cache-based**	Incoherent Cache-based
	S/W	(Impractial)	**Steaming Memory**

This work focuses on the two highlighted options: Coherent cache-based and streaming memory. The third practical option, incoherent cache-based, is briefly discussed in Section 7.

2.1 Coherent Cache-based Model

With the cache-based model [Culler et al. 1999], the only directly addressable storage is the off-chip memory. All on-chip storage is used for caches with hardware management of locality and communication. As cores perform memory accesses, the caches attempt to capture the application's working set by fetching or replacing data at the granularity of cache blocks. The cores communicate implicitly through loads and stores to the single memory image. Because many caches may store copies of a specific address, it is necessary to query multiple caches on load and store requests and potentially invalidate entries to keep caches coherent. A coherence protocol, such as MESI, minimizes the cases when remote cache lookups are necessary. Remote lookups can be distributed through a broadcast mechanism or by first consulting a directory structure. The cores synchronize using atomic operations such as compare-and-swap. Cache-based systems must also provide event-ordering guarantees within and across cores following some consistency model [Adve and Gharachorloo 1996]. Caching, coherence, synchronization, and consistency are implemented in hardware.

Coherent caching techniques were developed for board-level and cabinet-level systems (SMPs and DSMs), for which communication latency ranges from tens to hundreds of cycles. In CMPs, coherence signals travel within one chip, where latency is much lower and bandwidth is much higher. Consequently, even algorithms with nontrivial amounts of communication and synchronization can scale reasonably well. Moreover, the efficient design points for coherent caching in CMPs are likely to be different from those for SMP and DSM systems.

2.2 Streaming Memory Model

With streaming, the local memory for data in each core is a separately addressable structure called a *scratch-pad, local store*, or *stream register file*. We adopt the term *local store* in this work. Software is responsible for managing locality and communication across local stores. Software has full flexibility in placing frequently accessed data in local stores with respect to location, granularity, replacement policy, allocation policy, and even the number of copies. For applications with statically analyzable access patterns, software can exploit this flexibility to minimize communication and overlap it with useful computation in the best possible application-specific way. Data are communicated between local stores or to and from off-chip memory using explicit DMA transfers. The cores can access their local stores as FIFO queues or as randomly indexed structures [Jayasena 2005]. The streaming model requires DMA engines, but no other special hardware support for coherence or consistency.

2.3 Qualitative Comparison

When considering the memory system alone, we find several ways in which cache-based and streaming memory systems may differ: bandwidth utilization, latency tolerance, performance, energy consumption, and cost. This section summarizes each of these differences in a qualitative manner.

2.3.1 *Off-chip Bandwidth and Local Memory Utilization.* The cache-based model leads to bandwidth and storage capacity waste on sparsely strided memory accesses. In the absence of spatial locality, manipulating data at the granularity of wide cache lines is wasteful. Streaming memory systems, by virtue of strided scatter and gather DMA transfers, can use the minimum memory channel bandwidth necessary to deliver data, and also compact the data within the local store. Note, however, that memory and interconnect channels are typically optimized for block transfers and may not be bandwidth efficient for strided or scatter/gather accesses.

Caching can also waste off-chip bandwidth on unnecessary refills for output data. Because caches often use write-allocate policies, store misses force memory reads before the data are overwritten in the cache. If an application has disjoint input and output streams, the refills may waste a significant percentage of bandwidth. Similarly, caching can waste bandwidth on write-backs of dead temporary data. A streaming system does not suffer from these problems, as the output and temporary buffers are managed explicitly. Output data are sent off-chip without refills, and dead temporary data can be ignored, as they are not mapped to off-chip addresses. To mitigate the refill problem, cache-based systems can use a no-write-allocate policy. In this case, it is necessary to group store data in write buffers before forwarding them to memory in order to avoid wasting bandwidth on narrow writes [Andrews and Backer 2005]. Another approach is to use cache control instructions, such as "Prepare For Store," [MIPS32 2001] that instruct the cache to allocate a cache line but avoid retrieval of the old values from memory. Similarly, temporary data can be marked invalid at the end of a computation [Chiueh 1993; Wang et al. 2002]. In any case, software must determine when to use these mechanisms.

Streaming systems may also waste bandwidth and storage capacity on programs with statically unpredictable, irregular data patterns. A streaming system can sometimes cope with these patterns by fetching a superset of the needed input data. Alternatively, at the cost of enduring long latencies, the system could use a DMA transfer to collect required data on demand from main memory before each computational task. For programs that operate on overlapping blocks or graph structures with multiple references to the same data, the streaming system may naively re-fetch data. This can be avoided through increased address generation complexity or software caching. Finally, for applications that fetch a block and update some of its elements in-place, a streaming system will often write back the whole block to memory at the end of the computation, even if some data were not updated. In contrast to all of these scenarios, cache-based systems perform load and store accesses on demand and, hence, only move cache lines as required. They may even search for copies of the required data in other on-chip caches before going off-chip.

2.3.2 *Latency Tolerance.* The cache-based model is traditionally reactive, meaning that a miss must occur before a fetch is triggered. Memory latency can be hidden using hardware prefetching techniques, which detect repetitive access patterns and issue memory accesses ahead of time, or proactive software prefetching. In practice, the DMA transfers in the streaming memory model are an efficient and accurate form of software prefetching. They can hide a significant amount of latency, especially if double-buffering is used. Unlike hardware prefetching, which requires a few misses before a pattern is detected (microscopic view), a DMA access can start arbitrarily early (macroscopic view) and can capture both regular and irregular (scatter/gather) accesses.

2.3.3 *Performance.* From the discussion so far, one can conclude that streaming memory may have a performance or scaling advantage for regular applications, due to potentially better latency tolerance and better usage of off-chip bandwidth or local storage. These advantages are important only if latency, bandwidth, or local storage capacity are significant bottlenecks to begin with. For example, reducing the number of misses is unimportant for a computationally intensive application that already has very good locality. In the event that an application is bandwidth-bound, latency tolerance measures will be ineffective. A drawback for streaming, even with regular code, is that it often has to execute additional instructions to set up DMA transfers. For applications with unpredictable data access patterns or control flow, a streaming system may execute more instructions than that of a cache-based system to produce predictable patterns or to use the local store to emulate a software cache.

2.3.4 *Energy Consumption.* Any performance advantage also translates to an energy advantage, as it allows us to turn off the system early or scale down its power supply and clock frequency. Streaming accesses to the first-level storage eliminate the energy overhead of caches (tag access and tag comparison). The cache-based model consumes additional energy for on-chip coherence traffic, snoop requests or directory lookups. Moreover, efficient use of the available off-chip bandwidth by either of the two models (through fewer transfers or messages) reduces the energy consumption by the interconnect network and main memory.

2.3.5 *Complexity and Cost.* It is difficult to make accurate estimates of hardware cost without comparable implementations. The hardware for the cache-based model is generally more complex to design and verify, as coherence, synchronization, consistency, and prefetching interact in subtle ways. Still, reuse across server, desktop, and embedded CMP designs can significantly reduce such costs. On the other hand, streaming passes the complexity to software, the compiler, and/or the programmer. For applications in the synchronous data-flow, DSP, or dense matrix domains, it is often straight forward to express a streaming algorithm. For other applications, it is nontrivial, and a single good algorithm is often a research contribution in itself [Drake et al. 2006; Foley and Sugerman 2005]. Finally, complexity and cost must also be considered with

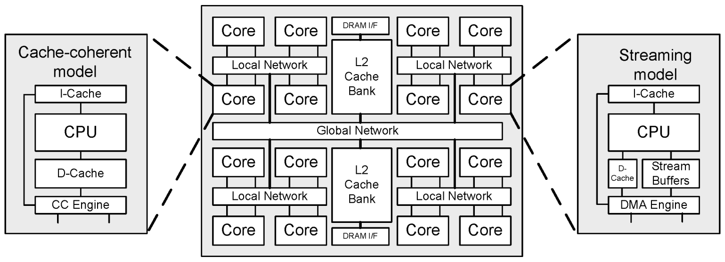


Fig. 1. The architecture of the CMP system with up to 16 processors. The core organizations for the cache-based and streaming models are shown on the left and right side respectively.

scaling in mind. A memory model has an advantage if it allows efficient use of more cores without the need for disproportional increases in bandwidth or some other resource in the memory system.

3. CMP ARCHITECTURE FRAMEWORK

We compare the two models using the CMP architecture shown in Figure 1. There are numerous design parameters in a CMP system, and evaluating the two models under all possible combinations is infeasible. Hence, we start with a baseline system that represents a practical design point and vary only the key parameters that interact significantly with the on-chip memory system.

3.1 Baseline Architecture

Our CMP design is based on in-order processors similar to Piranha [Barroso et al. 2000], RAW [Taylor et al. 2004], Ultrasparc T1 [Kongetira 2004], and Xbox360 [Andrews and Backer 2005]. Such CMPs have been shown to be efficient for multimedia, communications, and throughput computing workloads as they provide high compute density without the area and power overhead of out-of-order processors [Davis et al. 2005]. We use the Tensilica Xtensa LX, 3-way VLIW core [Jani et al. 2004]. Tensilica cores have been used in several embedded CMP designs, including the 188-core Cisco Metro router chip [Eatherton 2005]. We also performed experiments with a single-issue Xtensa core that produced similar results for the memory model comparison. The VLIW core is consistently faster by 1.6x to 2x for the applications we studied. The core has three slots per instruction, with up to two slots for floating-point operations and up to one slot for loads and stores. Due to time constraints, we do not use the Tensilica SIMD extensions at this point. Nevertheless, Section 5.3 includes an experiment that evaluates the efficiency of each model with processors that provide higher computational throughput. Each processor has a 16-KByte, 1-way set-associative instruction cache. The fixed amount of local data storage is used differently in each memory model.

We explore systems with 1 to 16 cores using a hierarchical interconnect similar to that suggested by [Kumar et al. 2005]. We group cores in clusters of four with a wide, bidirectional bus (*local network*) providing the necessary interconnect. The cluster structure allows for fast communication between neighboring

Table II. Parameters for the CMP System

Cache-Coherent Model (CC)

Data Storage	32KB, 2-way associative cache, 32-byte blocks, 1 port Hardware stream prefetcher 16 MSHRs

Streaming Model (STR)

Data Storage	24KB local store, 1 port 8KB, 2-way associative cache, 32-byte blocks, 1 port DMA engine with 16 outstanding accesses

Common Parameters

Core	1, 2, 4, 8, or **16** Tensilica LX cores, 3-way VLIW, 7-stage **800MHz**, 1.6GHz, 3.2GHz, or 6.4GHz clock frequency 2 FPUs, 2 integer units, 1 load/store unit
I-cache	16KB, 2-way associative cache, 32-byte blocks, 1 port
Local Network	bidirectional bus, 32 bytes wide, 2 cycle lat. (after arb.)
Global Crossbar	1 input and output port per cluster or L2 bank, 16 bytes wide, 2.5ns latency (pipelined)
L2-cache	**512KB**, 16-way set associative, 1 port, 2.2ns access latency, non-inclusive
DRAM	One memory channel at 1.6GB/s, **3.2GB/s**, 6.4GB/s, or 12.8GB/s; 70ns random access latency

For parameters that vary, we denote the default value in bold. Latencies are for a 90nm CMOS process.

cores. If threads are mapped intelligently, the intracluster bus will handle most of the core-to-core communication. A global crossbar connects the clusters to the second-level storage. There is buffering at all interfaces to tolerate arbitration latencies and ensure efficient bandwidth use at each level. The hierarchical interconnect provides sufficient bandwidth, while avoiding the bottlenecks of long wires and centralized arbitration [Kumar et al. 2005]. Finally, the secondary storage communicates to off-chip memory through some number of memory channels.

Table II presents the parameters of the CMP system. We vary the number of cores, the core clock frequency, the available off-chip bandwidth, and the degree of hardware prefetching. We keep the capacity of the first-level data-storage constant.

3.2 Cache-based Implementation

For the cache-based model, the first-level data storage in each core is organized as a 32-KB, 2-way set-associative data cache. The second-level cache is a 512-KB, 16-way set-associative cache. Both caches use a write-back, write-allocate policy. Coherence is maintained using the MESI write-invalidate protocol. Coherence requests are propagated in steps over the hierarchical interconnect. First, they are broadcast to other processors within the cluster. If the cache-line is not available or the request cannot be satisfied within one cluster (e.g., upgrade request), it is broadcast to all other clusters as well. Snooping requests from other cores occupy the data cache for one cycle, forcing the core to stall if it tries to do a load/store access in the same cycle. Each core includes a store-buffer that allows loads to bypass store misses. As a result, the consistency model is weak. Because the processors are in-order, it is easy to provide sufficient MSHRs for the maximum possible number of concurrent misses.

Each core additionally includes a hardware stream-based prefetch engine that places data directly in the L1 cache. Modeled after the tagged prefetcher described in [VanderWiel and Lilja 2000], the prefetcher keeps a history of the last eight cache misses for identifying sequential accesses, runs a configurable number of cache lines ahead of the latest cache miss, and tracks four separate access streams. Our experiments include hardware prefetching only when explicitly stated.

We use POSIX threads to manually parallelize and tune applications [Lewis and Berg 1998]. The applications used are regular and use locks to implement efficient task-queues and barriers to synchronize SPMD code. Higher-level programming models, such as OpenMP, are also applicable to these applications.

3.3 Streaming Implementation

For the streaming model, the first-level data storage in each core is split between a 24-KB local store and an 8-KB cache. The small cache is used for stack data and global variables. It is particularly useful for the sequential portions of the code and helps simplify the programming and compilation of the streaming portion as well. The 24-KB local store is indexed as a random access memory. Our implementation also provides hardware support for FIFO accesses to the local store, but we did not use this feature with any of our applications. Each core has a DMA engine that supports sequential, strided, and indexed transfers, as well as command queuing. At any point, each DMA engine may have up to sixteen 32-byte outstanding accesses. This matches the number of MSHRs we modeled in our L1 cache controllers.

The local store is effectively smaller than a 24-KB cache because it has no tag or control bits overhead. We do not raise the size of the local store, since the small increment (2KB) does not make a difference for our applications. Still, the energy consumption model accounts correctly for the reduced capacity. The secondary storage is again organized as a 16-way set-associative cache. L2 caches are useful with stream processors, as they capture long-term reuse patterns and avoid expensive accesses to main memory [Sankaralingam 2004; Dally et al. 2003]. The L2 cache avoids refills on write misses when DMA transfers overwrite entire lines.

We developed streaming code using a simple library for DMA transfers within threaded C code. We manually applied the proper blocking transformation and double-buffering in order to overlap DMA transfers with useful computation. We also run multiple computational kernels on each data block to benefit from producer-consumer locality without additional memory accesses or write-backs for intermediate results. Higher-level stream programming models should be applicable to most of our applications [Gordon et al. 2002; Fatahalian et al. 2006]. In some cases, the DMA library uses a scheduling thread that queues pending transfers. We avoid taking up a whole core for this thread by multiplexing it with an application thread.

The high-level DMA interface resembles an asynchronous `memcpy` routine. The programmer specifies a DMA engine identifier, a source address, a destination address, and a transfer size. Strided transfers are additionally specified by a transfer stride and transfer count. Indexed transfers are additionally specified by a transfer count and an address vector for the source address or destination address, depending on whether the transfer is a scatter or gather. DMA engine status (i.e., transfer completion) can be queried by decoding a memory-mapped status register. The low-level DMA interface is implemented with memory-mapped device control registers. Consequently, it takes only a few cycles to program a DMA engine and initiate a transfer.

4. METHODOLOGY

4.1 Simulation and Energy Modeling

We used Tensilica's [2007] modeling tools to construct a CMP simulator for both memory models. The simulator captures all stall and contention events in the core pipeline and the memory system. Table II summarizes the major system characteristics and the parameters we varied for this study. The default values are shown in bold. The applications were compiled with Tensilica's optimizing compiler at the -O3 optimization level. We fast-forward over the initialization for each application but simulate the rest to completion, excluding 179.art for which we measure 10 invocations of the train_match function.

We also developed an energy model for the architecture in a 90nm CMOS process (1.0V power supply). For the cores, the model combines usage statistics (instruction mix, functional unit utilization, stalls and idle cycles, etc.) with energy data from the layout of actual Tensilica designs at 600MHz in 90nm. The energy consumed by on-chip memory structures is calculated using CACTI 4.1 [Tarjan et al. 2006], which includes a leakage power model and improved circuit models compared to CACTI 3. Interconnect energy is calculated based on our measured activity statistics and scaled power measurements from [Ho et al. 2003]. The energy consumption for off-chip DRAM is derived from DRAMsim [Wang et al. 2005]. We model the effect of leakage and clock gating on energy at all levels of the model.

4.2 Applications

Table III presents the set of applications used for this study. They represent applications from the multimedia, graphics, physical simulation, DSP, and data management domains. Such applications have been used to evaluate and motivate the development of streaming architectures. MPEG-2, H.264, Raytracer, JPEG, and Stereo Depth Extraction are compute-intensive applications and show exceptionally good cache performance despite their large datasets. They exhibit good spatial or temporal locality and have enough computation per data element to amortize the penalty for any misses. FEM is a scientific application, but has about the same compute intensity as multimedia applications. The remaining applications—Bitonic Sort, Merge Sort, FIR, and 179.art—perform a

Table III. Memory Characteristics of the Applications Measured on the Cache-based Model Using 16 Cores Running at 800MHz

Application	Input Dataset
MPEG-2 Encoder [MPEG Software Simulation Group]	10 CIF frames (Foreman sequence)
H.264 Encoder [ITU-T Rec. H.264]	10 CIF frames (Foreman sequence)
KD-tree Raytracer [Havran 2002]	Glassner scene, 16371 triangles, 128x128 rays
JPEG Encoder [Independent JPEG Group 1998]	128 PPMs of various sizes
JPEG Decoder [Independent JPEG Group 1998]	128 JPGs of various sizes
Stereo Depth Extraction	3 CIF image pairs
2D Finite Element Method (FEM)	5006 cell mesh, 7663 edges
Finite Impulse Response filter (FIR)	2^{20} 32-bit samples
CFP2000 179.art (image recognition)	SPEC reference dataset
Bitonic Sort	2^{19} 32-bit keys (2 MB)
Merge Sort	2^{19} 32-bit keys (2 MB)

Application	L1 D-Miss Rate	L2 D-Miss Rate	Instr. per L1 D-Miss	Cycles per L2 D-Miss	Off-chip B/W
MPEG-2 Encoder	0.58%	85.3%	324.8	135.4	292.4 MB/s
H.264 Encoder	0.06%	30.8%	3705.5	4225.9	10.8 MB/s
KD-tree Raytracer	1.06%	98.9%	256.3	654.6	45.1 MB/s
JPEG Encoder	0.40%	72.9%	577.1	84.2	402.2 MB/s
JPEG Decoder	0.58%	76.2%	352.9	44.9	1059.2 MB/s
Stereo Depth Extraction	0.03%	46.1%	8662.5	3995.3	11.4 MB/s
2D FEM	0.60%	86.2%	368.8	55.5	587.9 MB/s
FIR filter	0.63%	99.8%	214.6	20.4	1839.1 MB/s
179.art	1.79%	7.4%	150.1	230.9	227.7 MB/s
Bitonic Sort	2.22%	98.2%	140.9	26.1	1594.2 MB/s
Merge Sort	3.98%	99.7%	71.1	33.7	1167.8 MB/s

relatively small computation on each input element. They require considerably higher off-chip bandwidth and are sensitive to memory latency.

We manually optimized both versions of each application to eliminate bottlenecks and schedule its parallelism in the best possible way. Whenever appropriate, we applied the same data-locality optimizations (i.e., blocking, producer-consumer) to both models. In Section 6, we explore the impact of data-locality optimizations. The following is a brief description of how each application was parallelized.

MPEG-2 and *H.264* are parallelized at the macroblock level. Both dynamically assign macroblocks to cores using a task queue. Macroblocks within a single frame are entirely data-parallel in MPEG-2. In H.264, we schedule the processing of dependent macroblocks so as to minimize the length of the critical execution path, similar to [Chen et al. 2006]. With the CIF resolution video frames we encode for this study, the macroblock parallelism available in H.264 is limited. *Stereo Depth Extraction* is parallelized by dividing input frames into 32x32 blocks and statically assigning them to processors.

KD-tree Raytracer is parallelized across camera rays. We assign rays to processors in chunks to improve locality. Our streaming version reads the KD-tree from the cache instead of streaming it with a DMA controller. *JPEG Encode* and *JPEG Decode* are parallelized across input images, in a manner similar to that done by an image thumbnail browser. Note that Encode reads a lot of data but outputs little; Decode behaves in the opposite way. The *Finite Element Method (FEM)* is parallelized across mesh cells. The *FIR filter* has 16 taps and is parallelized across long strips of samples. *179.art* is parallelized across F1 neurons; this application is composed of several data-parallel vector operations and reductions between which we place barriers.

Merge Sort and *Bitonic Sort* are parallelized across subarrays of a large input array. The processors first sort chunks of 4,096 keys in parallel using quicksort. Then, sorted chunks are merged or sorted until the full array is sorted. Merge Sort gradually reduces in parallelism as it progress, whereas Bitonic Sort retains full parallelism for its duration. Merge Sort alternates writing output sublists to two buffer arrays, while Bitonic Sort operates on the list *in situ*.

5. EVALUATION

Our evaluation starts with a comparison of the streaming system to the baseline cache-based system without prefetching or other enhancements (Sections 5.1 and 5.2). We then study the bandwidth consumption and latency tolerance of the two systems (Section 5.3), and continue by evaluating means to enhance the performance of caching systems (Sections 5.4 and 5.5). Finally, we consider the impact of secondary storage in Section 5.6.

5.1 Performance Comparison

Figure 2 presents the execution time for the coherent cache-based (CC) and streaming (STR) models as we vary the number of 800MHz cores from 2 to 16. We normalize to the execution time of the sequential run with the cache-based system. The components of execution time are: useful execution (including fetch and nonmemory pipeline stalls), synchronization (lock, barrier, wait for DMA), and stalls for data (caches). Lower bars are better. The cache-based results assume no hardware prefetching.

For 7 out of 11 applications (MPEG-2, H.264, Depth, Raytracing, FEM, JPEG Encode, and Decode), the two models perform almost identically for all processor counts. These programs perform a significant computation on each data element fetched and can be classified as compute-bound. Both caches and local stores capture their locality patterns equally well.

The remaining applications (179.art, FIR, Merge Sort, and Bitonic Sort) are data-bound and reveal some interesting differences between the two models. The cache-based versions stall regularly due to cache misses. Our streaming experiments eliminate many of these stalls using double-buffering (macroscopic prefetching). This is not the case for Bitonic Sort, because off-chip bandwidth is saturated at high processor counts. Bitonic Sort is an in-place sorting algorithm, and it is often the case that sublists are moderately in-order and elements do not need to be swapped, and consequently do not need to be written back. The cache-based system naturally discovers this behavior, while the streaming memory

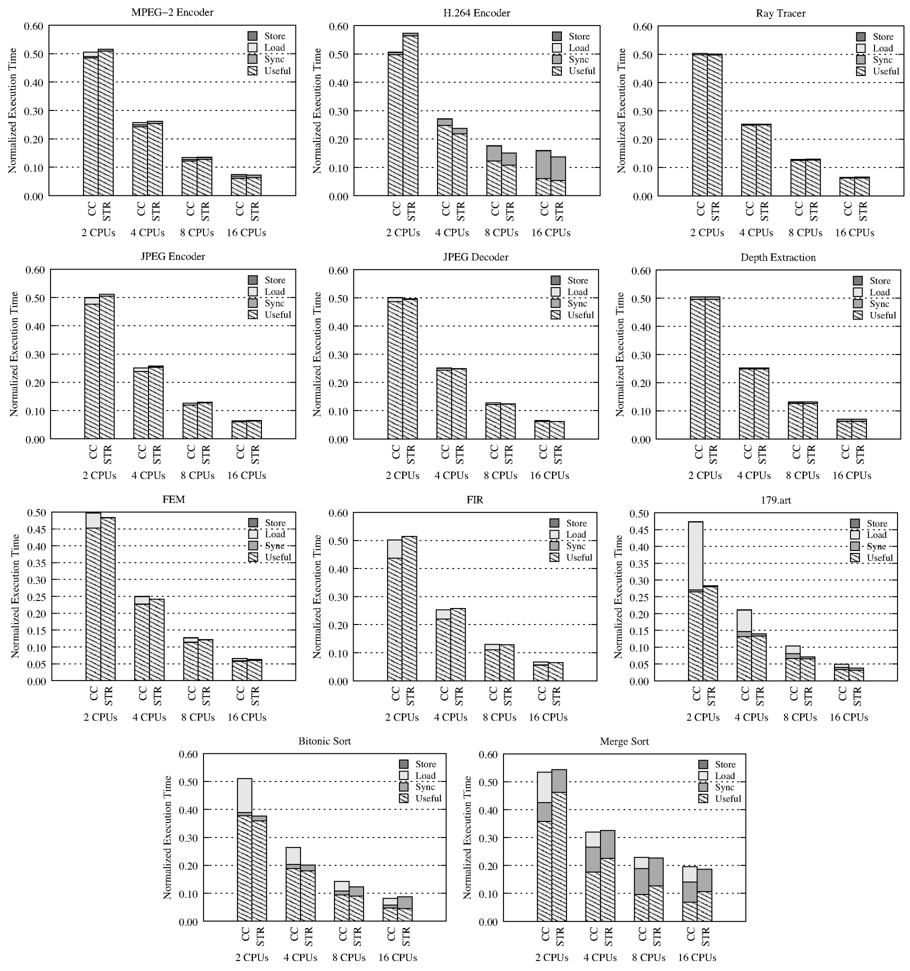


Fig. 2. Execution times for the two memory models as the number of cores is increased, normalized to a single caching core.

system writes the unmodified data back to main memory anyway. H.264 and Merge Sort have synchronization stalls with both models due to limited parallelism.

There are subtle differences in the useful cycles of some applications. FIR executes 14% more instructions in the streaming model than the caching model because of the management of DMA transfers (128 elements per transfer). In the streaming Merge Sort, the inner loop executes extra comparisons to check if an output buffer is full and needs to be drained to main memory, whereas the cache-based variant freely writes data sequentially. Even though double-buffering eliminates all data stalls, the application runs longer because of its higher instruction count. The streaming H.264 takes advantage of some boundary-condition optimizations that proved difficult in the cache-based variant. This resulted in a slight reduction in instruction count when streaming.

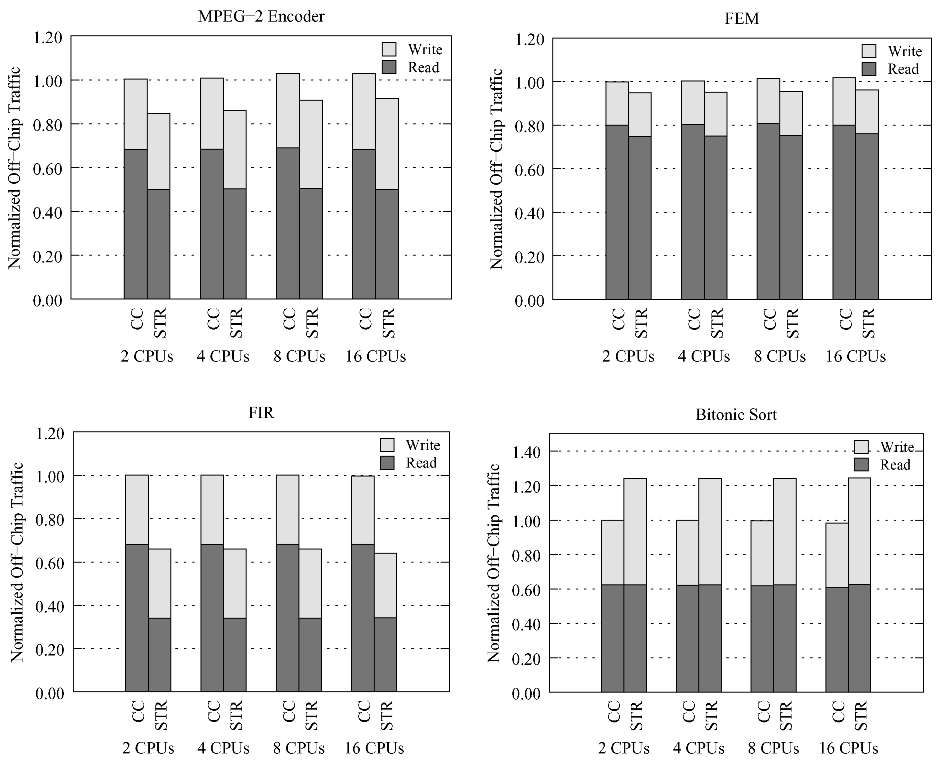


Fig. 3. Off-chip traffic for the two memory models as the numbers of cores is increased, normalized to a single caching core.

MPEG-2 suffers a moderate number of instruction cache misses due the cache's limited size.

Overall, Figure 2 shows that neither model has a consistent performance advantage. Even without prefetching, the cache-based model performs similarly to the streaming one, and sometimes it is actually faster (Bitonic Sort for 16 cores). Both models use data efficiently, and the fundamental parallelism available in these applications is not affected by how data are moved. Although the differences in performance are small, there is a larger variation in off-chip bandwidth utilization of the two models. This accounts for the slowdown of the streaming Bitonic Sort with 16 cores. Figure 3 shows that each model has an advantage in some situations. We explore bandwidth more thoroughly in Section 5.4.

5.2 Energy Comparison

Figure 4 presents energy consumption for the two models running FEM, MPEG-2, FIR, and Bitonic Sort. We normalize to the energy consumption of a single caching core for each application. Each bar indicates the energy consumed by the cores, the caches and local stores, the on-chip network, the second-level cache, and the main memory. The numbers include both static and dynamic power. Lower bars are better. In contrast to performance scaling, energy

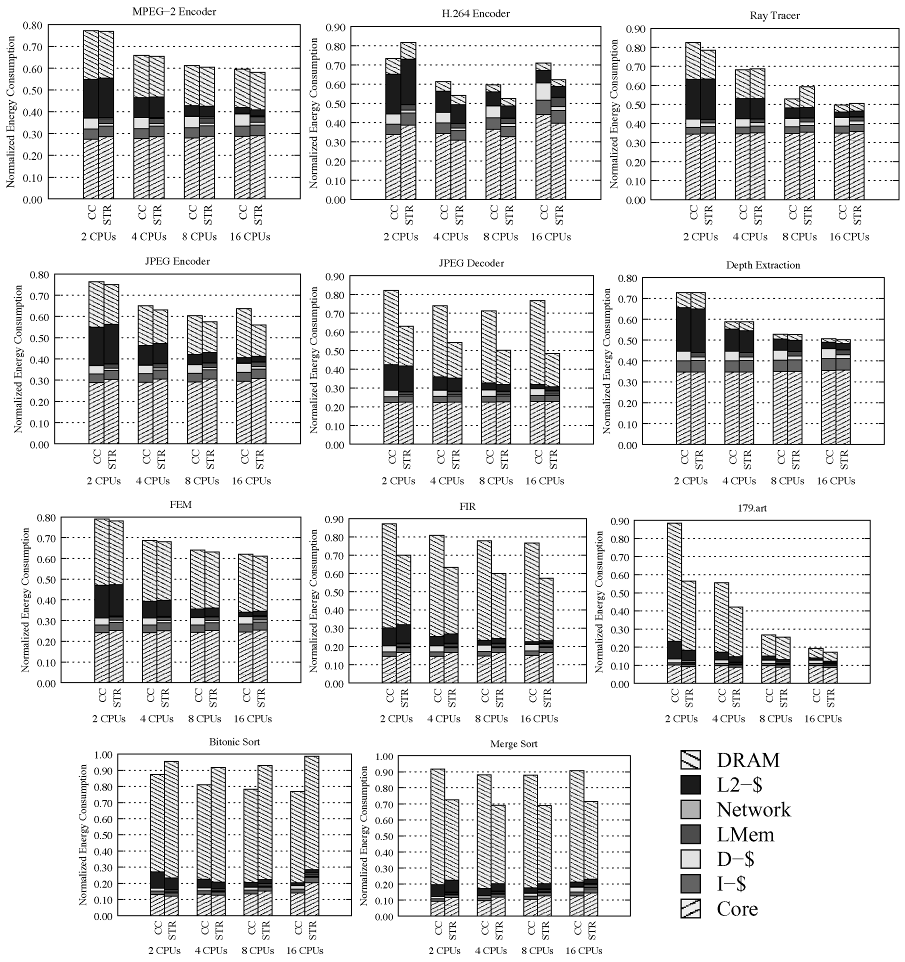


Fig. 4. Energy consumption for the two memory models as the number of cores is increased, normalized to a single caching core.

consumption does not always improve with more cores, since the amount of hardware used to run the application increases.

For 5 out of 11 applications (JPEG Encode, JPEG Decode, FIR, 179.art, and Merge Sort), streaming consistently consumes less energy than cache-coherence, typically 10% to 25%. The energy differential in nearly every case comes from the DRAM system. This can be observed in the correlation between Figures 3 and 4. Specifically, the streaming applications typically transfer fewer bytes from main memory, often through the elimination of superfluous refills for output-only data. The opposite is true for our streaming Bitonic Sort, which tends to communicate more data with main memory than the caching version due to the write-back of unmodified data. For applications where there is little bandwidth difference between the two models (such as FEM) or the computational intensity is very high (such as Depth), the difference in energy consumption is insignificant.

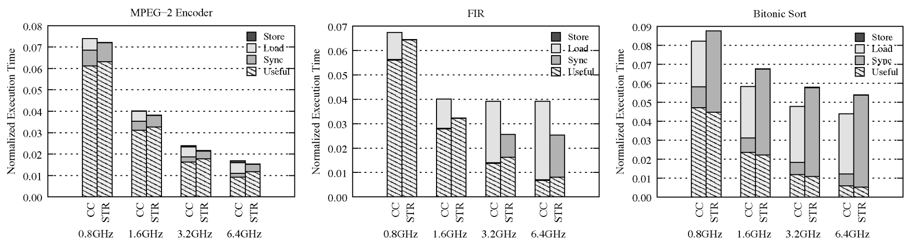


Fig. 5. Normalized execution time as the computation rate of processor cores is increased (16 cores).

We expected to see a greater difference between the local store and L1 data cache, but it never materialized. Since our applications are data-parallel and rarely share data, the energy cost of an average cache miss is dominated by the off-chip DRAM access rather than the modest tag broadcast and lookup. Hence, the peraccess energy savings by eliminating tag lookups in the streaming system made little impact on the total energy footprint of the system.

5.3 Increased Computational Throughput

Up to now, the results assume 800MHz cores, which are reasonable for embedded CMPs for consumer applications. To explore the efficiency of the two memory models as the computational throughput of the processor is increased, we vary the clock frequency of the cores while keeping constant the bandwidth and latency in the on-chip networks, L2 cache, and off-chip memory. In some sense, the higher clock frequencies tell us in general what would happen with more powerful processors that use SIMD units, out-of-order schemes, higher clock frequency, or a combination. For example, the 6.4GHz configuration can be representative of the performance of an 800MHz processor that uses 4- to 8-wide SIMD instructions. The experiment was performed with 16 cores to stress scaling and increase the system's sensitivity to both memory latency and memory bandwidth.

Applications with significant data reuse, such as H.264 and Depth, show no sensitivity to this experiment and perform equally well on both systems. Figure 5 shows the results for some of the applications that are affected by computational scaling. These applications fall into one of two categories: bandwidth-sensitive or latency-sensitive. Latency-sensitive programs, like MPEG-2 Encoding, perform a relative large degree of computation between off-chip memory accesses (hundreds of instructions). While the higher core frequency shortens these computations, it does not reduce the amount of time (in nanoseconds, not cycles) required to fetch the data in between computations. The macroscopic prefetching in the streaming system can tolerate a significant percentage of the memory latency. Hence, at 6.4GHz, the streaming MPEG-2 Encoder is 9% faster.

Bandwidth-sensitive applications, such as FIR and Bitonic Sort, eventually saturate the available off-chip bandwidth. Beyond that point, further increases in computational throughput do not improve overall performance. For FIR, the cache-based system saturates before the streaming system due to the superfluous refills on store misses to output-only data. At the highest computational

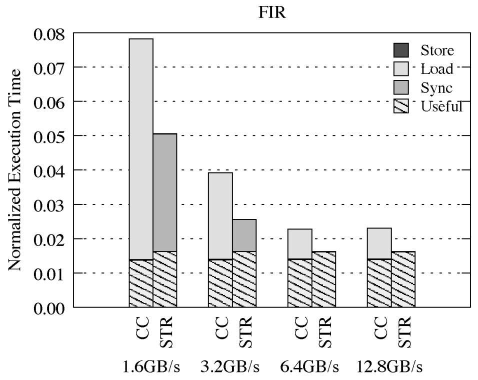


Fig. 6. The effect of increased off-chip bandwidth on FIR performance. Measured on 16 cores at 3.2GHz.

throughput, the streaming system performs 36% faster. For Bitonic Sort, the streaming version saturates first, because it performs more writes than the cache-based version (as described in Section 5.1). This gives the cache-based version a 19% performance advantage.

5.4 Mitigating Latency and Bandwidth Issues

The previous section indicates that when a large number of cores with high computational throughput are used, the cache-based model faces latency and bandwidth issues with certain applications. To characterize these inefficiencies, we performed experiments with increased off-chip bandwidth and hardware prefetching.

Figure 6 shows the impact of increasing the available off-chip bandwidth for FIR. This can be achieved by using higher frequency DRAM (e.g., DDR2, DDR3, XDR) or multiple memory channels. With more bandwidth available, the effect of superfluous refills is significantly reduced, and the cache-based system performs nearly as well as the streaming one. When hardware prefetching is introduced at 12.8GB/s, load stalls are reduced to 3% of the total execution time. However, the additional off-chip bandwidth does not close the energy gap for this application. An energy-efficient solution for the cache-based system is to use a nonallocating write policy, which we explore in Section 5.5.

For Merge Sort and 179.art (Figure 7), hardware prefetching significantly improves the latency tolerance of the cache-based systems; data stalls are virtually eliminated. This is not to say that we never observed data stalls—at 16 cores, the cache-based Merge Sort saturates the memory channel due to superfluous refills—but that a small degree of prefetching is sufficient to hide over 200 cycles of memory latency.

5.5 Mitigating Superfluous Refills

For some applications, the cache-based system uses more off-chip bandwidth (and consequently energy) because of superfluous refills for output-only data. This disadvantage can be addressed by using a non-write-allocate policy for

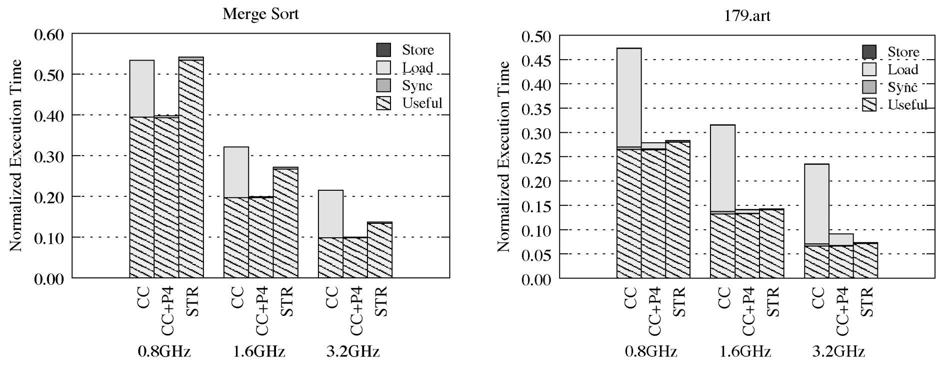


Fig. 7. The effect of hardware prefetching on performance. P4 refers to a prefetch depth of 4. Measured on two cores at the indicated clock rate with a 12.8GB/s memory channel.

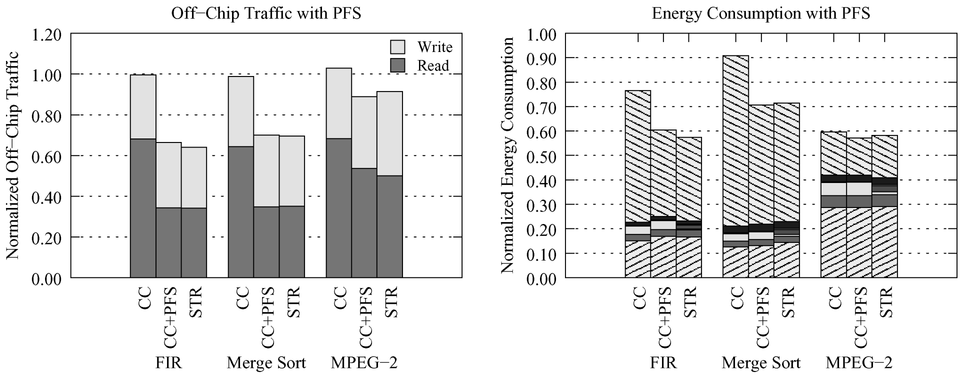


Fig. 8. The effect of "Prepare for Store" (PFS) instructions on the off-chip traffic and energy consumption for the cache-based system, normalized to a single caching core. Measured with 16 cores at 800MHz.

output-only data streams. We mimic nonallocating stores by using an instruction similar to the MIPS32 "Prepare for Store" (PFS) instruction [MIPS32 2001]. PFS allocates and validates a cache line without refilling it, and, it is fully coherent. The results for FIR, Mergesort, and MPEG-2 are shown in Figure 8. For each application, the elimination of superfluous refills brings the memory traffic and energy consumption of the cache-based model into parity with the streaming model. For MPEG-2, the memory traffic, due to write misses, was reduced 56% compared to the cache-based application without PFS.

Note that a full hardware implementation of a non-write-allocate cache policy, along with the necessary write-gathering buffer, might perform better than PFS, as it would also eliminate cache *replacements* due to output-only data.

5.6 The Importance of Secondary Caches

Figure 9 studies the sensitivity of the two models to the capacity of the L2 cache. The L2 cache has little impact on performance for both models as the critical working sets of the applications tend to either easily fit in the L1 cache or not fit in even the L2 cache. On the other hand, we noticed increases in the

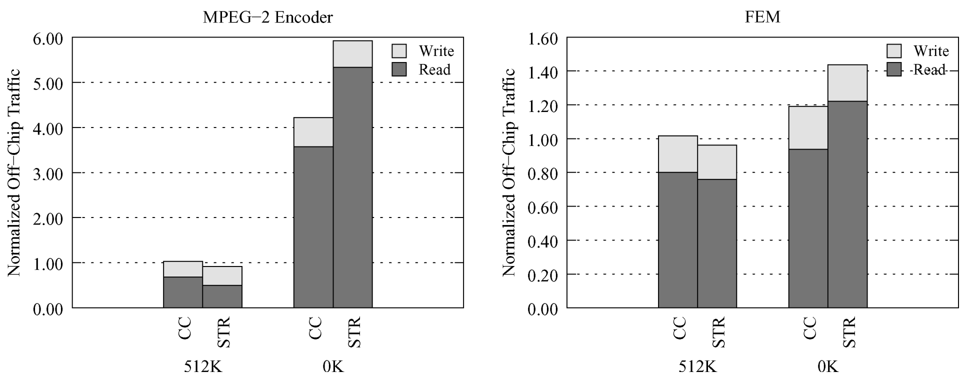


Fig. 9. Off-chip accesses with and without an L2 cache for MPEG-2 and FEM (16 cores), normalized to a single caching core with L2 enabled. No prefetching.

energy consumption of the two models when we eliminated the L2 completely for MPEG-2 and H.264. For these applications, the multiple cores operate on overlapping regions of nearby macroblocks. Hence, there is spatial locality in data accesses captured by the L2 for both models. When the L2 is eliminated, the cache-based model captures some of this locality in the L1 caches. With the streaming model, all accesses go to off-chip memory. FEM, which exhibits redundant gathers on nearby mesh cells, also shows an increase in off-chip accesses when the L2 is eliminated, where as the cache-based model keeps redundant references local.

In our experiments, these redundant accesses had little impact on performance. On the other hand, any reduction in off-chip traffic will improve energy efficiency. It is possible for streaming architectures to overcome these inefficiencies at the cost of programmer or compiler effort. For example, redundant fetches can be mitigated by the repositioning of data within the local store, increasing address generation complexity, or by sorting indexed gather indices to more easily identify redundant addresses. Many such techniques are thoroughly evaluated in the context of scientific applications on streaming memory systems in [Erez et al. 2007]. The presence of an L2 cache can mitigate this effort.

5.7 Summary

This section has presented the performance, bandwidth consumption, and energy consumption of cache-based and streaming memory systems on CMPs of up to 16 cores. For most applications, the streaming and cache-based memory systems perform and scale equally well. For applications with significant disparities between memory systems, the application of prefetching and data locality management eliminates the difference.

6. STREAMING AS A PROGRAMMING MODEL

Our evaluation shows that the two memory models lead to similar performance and scaling. It is important to remember that we took advantage of streaming optimizations, such as blocking and locality-aware scheduling, on both memory

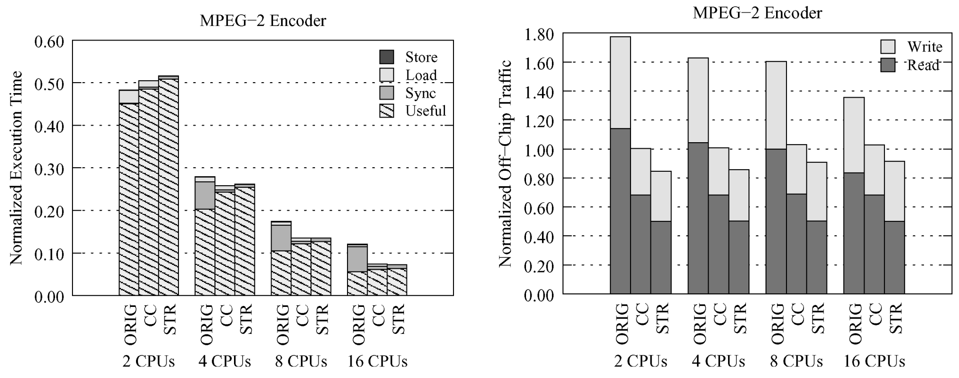


Fig. 10. The effect of stream programming optimizations on the performance and off-chip bandwidth of MPEG-2 at 800MHz.

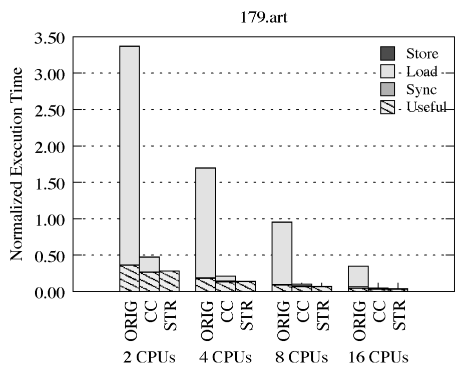
models. To illustrate this, it is educational to look at stream programming and its implications for CMP architecture.

6.1 Streaming on Cache-based Systems

Stream programming models encourage programmers to think about the locality, data movement, and storage capacity issues in their applications [Gordon et al. 2002; Fatahalian et al. 2006]. Although they do not necessarily require the programmer to manage these issues, the programmer structures the application in such a way that it is easy to reason about them. This exposed nature of a stream program is vitally important for streaming architectures, as it enables software or compiler management of data locality and asynchronous communication with architecturally visible on-chip memories. Despite its affinity for stream architectures, we find that stream programming is beneficial for cache-based architectures as well.

Figure 10 shows the importance of streaming optimizations in the cache-based MPEG-2 Encoder. The original parallel code from [Li et al. 2005] performs an application kernel on a whole video frame before the next kernel is invoked (i.e., Motion Estimation, DCT, Quantization). We restructured this code by hoisting the inner loops of several tasks into a single outer loop that calls each task in turn. In the optimized version, we execute all tasks on a block of a frame before moving to the next block. This also allowed us to condense a large temporary array into a small stack variable. The improved producer-consumer locality reduced write-backs from L1 caches by 60%. Data stalls are reduced by 41% at 6.4GHz, even without prefetching. Furthermore, improving the parallel efficiency of the application became a simple matter of scheduling a single data-parallel loop, which alone is responsible for a 40% performance improvement at 16 cores. However, instruction cache misses are notably increased in the streaming-optimized code.

For 179.art, we reorganized the main data structure in the cache-based version in the same way as we did for the streaming code. We were also able to replace several large temporary vectors with scalar values by merging several



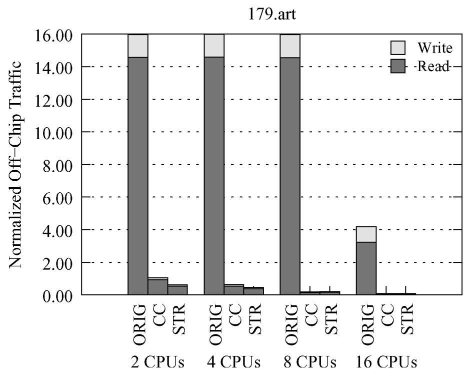


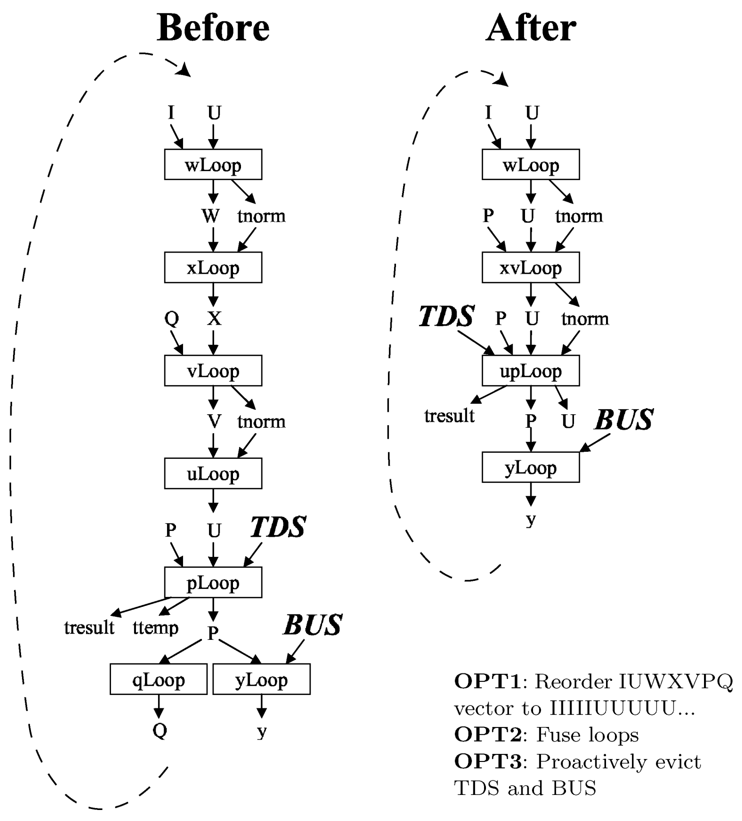
Fig. 11. The effect of stream programming optimizations on the performance and off-chip bandwidth of 179.art at 800MHz.

loops. These optimizations reduced the sparseness of 179.art's memory access pattern, improved both temporal and spatial locality, and allowed us to use prefetching effectively. As shown in Figure 11, the impact on performance is dramatic, even at small core counts (7x speedup). We discuss the optimizations for 179.art further in Section 6.2.

Overall, we observed performance, bandwidth, and energy benefits whenever stream programming optimizations were applied to our cache-based applications. This is not a novel result, since it is well known that locality optimizations, such as blocking and loop fusion [Lim et al. 2001], increase computational intensity and cache efficiency. However, stream programming models encourage users to write code that explicitly exposes an application's parallelism and data-access pattern, more often allowing such optimizations.

Our experience is that that stream programming is actually *easier* with the cache-based model rather than the streaming model. With streaming memory, the programmer or the compiler must orchestrate all data movement and positioning exactly right in order for the program to operate correctly and fast. This can be burdensome for irregular access patterns (overlapping blocks, search structures, unpredictable or data-dependent patterns, etc.), or for accesses that do not affect an application's performance. It can lead to additional instructions and memory references that reduce or eliminate streaming hardware's other advantages. With cache-based hardware, stream programming is just an issue of performance optimization. Even if the algorithm is not blocked exactly right, the caches will provide best-effort locality and communication management. Hence, the programmer or the compiler can focus on the most promising and most regular data structures instead of managing all data structures in a program.

Moreover, stream programming can address some of the coherence and consistency challenges when scaling cache-based CMPs to large numbers of cores. Since a streaming application typically operates in a data-parallel fashion on a sequence of data, there is little short-term communication or synchronization between processors. Communication is only necessary when



OPT1: Reorder IUWXVPQ vector to IIIIIUUUUU...
OPT2: Fuse loops
OPT3: Proactively evict TDS and BUS

Fig. 12. Inner loop sequence of train_match() in 179.art before and after streaming optimizations (producer-consumer). Vectors are labeled with capital letters and scalar values with lowercase letters. BUS and TDS are matrices that are more than 10x larger than each vector.

processors move from one independent set of input/output blocks to the next or reach a cycle in an application's data-flow graph. This observation may be increasingly important as CMPs grow, as it implies that less aggressive, coarser-grain, or lower-frequency mechanisms can be employed to keep caches coherent.

6.2 Application Study: 179.art

To illustrate the benefits from stream programming for cache-based systems, we will discuss in detail some of the optimizations for 179.art, a neural network simulator which is used to recognize objects in a thermal image. The application consists of two parts: training of the neural network and recognition of learned images. Both parts are very similar, as they do data-parallel vector operations followed by reductions.

The left side of Figure 12 shows the sequence of inner loops (vector operations) of the `train_match()` function and its outer loop. For example, `wLoop` iterates over elements of the `I` and `U` vectors and produces a vector `W` and a scalar `tnorm`, the result of a local reduction calculation, which must be shared between

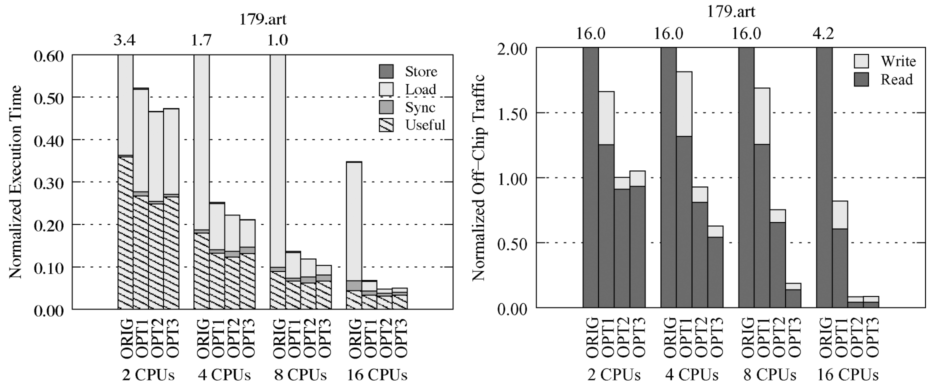
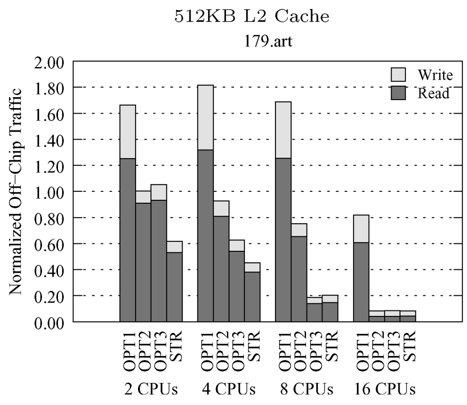


Fig. 13. The effect of stream programming optimizations the cache-based 179.art's performance and off-chip traffic. Measured at 800MHz.

processors. The original sequential version of this benchmark exhibits very low cache locality (35% L1 miss rate). This is because the main data structure of the application (`f1_layer`, is an array of structures which groups elements of vectors `I`, `U`, `W`, `X`, `V`, `P`, and `Q` together. Hence, consecutive elements of the same vector are not adjacent in memory.

To write 179.art as a stream program, we implemented the following optimizations. First, we restructured `f1_layer` so that consecutive elements of each vector are in sequential memory locations (**OPT1**). This dramatically improved spatial locality and reduced the L1 miss rate of the cache-based version to 10%. Second, we fused loops as shown in the right side of Figure 12, which eliminated several temporary vectors (**OPT2**). For example, after merging `xLoop` and `vLoop` into `xvLoop`, `X` becomes local to `xvLoop` and does not need to be written into memory. OPT2 embodies producer-consumer locality. This optimization does not necessarily improve performance but significantly reduces off-chip traffic and cache miss rate for cache-based systems. The last optimization (**OPT3**) was to carefully allocate and manage first-level storage for frequently used data. As the number of processors increases, a larger portion of the active working set can be held in first-level storage. To achieve this optimization on the cache-based system, we used a cache-control instruction to flush data that had little temporal locality. Figure 13 shows the impact of these optimizations on the performance, off-chip traffic, and energy consumption of the cache-based model. OPT1 leads to dramatic performance improvements across all processor counts. OPT2 and OPT3 do not affect performance significantly but reduce off-chip bandwidth requirements and energy consumption.

Figure 14 shows the difference in off-chip traffic between the cache-based versions and the optimized streaming version with the L2 cache enabled and disabled. OPT2 uses substantially more off-chip bandwidth than the streaming version because of cache conflicts. The introduction of invalidation hints in OPT3 reduces, but does not entirely eliminate, the disparity in both L2 cache scenarios.



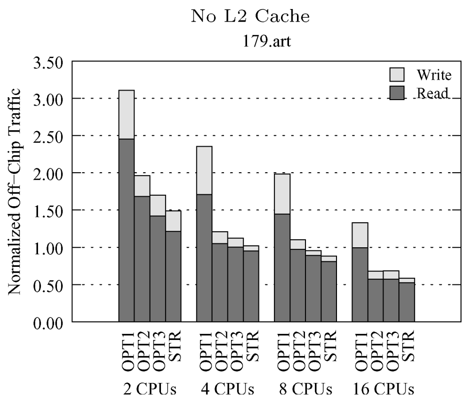


Fig. 14. Off-chip traffic of 179.art with 512KB L2 cache enabled and disabled. The difference in storage efficiency between the cache-based and streaming memory systems is most pronounced in the L2 cache.

7. DISCUSSION AND LIMITATIONS

It is important to recognize that our study has limitations. Our experiments focus on CMPs with 1 to 16 cores and uniform memory access. Some of the conclusions may not generalize to larger-scale CMPs with nonuniform memory access (NUMA). A large-scale, cache-based CMP, programmed in a locality-oblivious way, will undoubtedly suffer stalls due to long memory delays or excessive on-chip coherence traffic. We observe that the stream programming model may be able to addresses both limitations; it exposes the flow of data early enough that they can be prefetched, and motivates a far coarser-grained, lower-frequency coherence model.

This study does not address many important differences between streaming and cache-based memory systems. We do not compare the logic complexity of DMA engines or the load-store units in streaming memory systems to the cache-controller in cache-based memory systems. We also do not evaluate the efficiency of DRAM channel scheduling for each model. In general, we focused our efforts on the locality and programmability issues salient to all CMP systems.

Toward the goal of designing large CMPs that are still easy to program, a hybrid memory system that combines caching and software-managed memory concepts can mitigate efficiency challenges without exacerbating the difficulty of software development. For example, Gummaraju et al. [2007] extensively studied the addition of a “Stream Load/Store Unit,” which is optimized for generating bulk-transfer requests, to a general-purpose processor. Conversely, small, potentially incoherent caches in streaming memory systems could vastly simplify the use of static data structures with abundant temporal locality.

Other than the limits of scalability, we did not consider architectures that expose the streaming model all the way to the register file [Taylor et al. 2004] or applications without abundant data parallelism. We also did not consider changes to the pipeline of our cores, since that is precisely what makes it difficult to evaluate existing streaming memory processors compared to cache-based processors. Finally, our study was performed using general-purpose CMPs. A comparison between the two memory models for specialized CMPs remains an open issue. Despite these limitations, we believe this study's conclusions are important in terms of understanding the actual behavior of CMP memory system and motivating future research and development.

8. RELATED WORK

Several architectures [Ahn et al. 2004; Taylor et al. 2004; Gschwind et al. 2005; Machnicki 2005; Khailany et al. 2008] use streaming hardware with multimedia and scientific codes to get performance and energy benefits from software-managed memory hierarchies and regular control flow. There are also corresponding proposals for stream programming languages and compiler optimizations [Gordon et al. 2002; Fatahalian et al. 2006]. Such tools can reduce the burden on the programmer for explicit locality and communication management. In parallel, there are significant efforts to enhance cache-based systems with traffic filters [Moshovos 2005], replacement hints [Wang et al. 2002], or prefetching hints [Wang et al. 2003]. These enhancements target the same access patterns that streaming memory systems benefit from.

To the best of our knowledge, our prior work was the first direct comparison of the two memory models for CMP systems under a unified set of assumptions [Leverich et al. 2007]. Jayasena [2005] compared a stream register file to a single-level cache for a SIMD processor. He found that the stream register file provides performance and bandwidth advantages for applications with significant producer-consumer locality. Loghi and Poncino [2005] compared hardware cache coherence to not caching shared data at all for embedded CMPs with on-chip main memory. The ALP report [Li et al. 2005] evaluates multimedia codes on CMPs with streaming support. However, for all but one benchmark, streaming implied the use of enhanced SIMD instructions, not software-managed memory hierarchies. Suh et al. [2003] compared a streaming SIMD processor, a streaming CMP chip, a vector design, and a superscalar processor for DSP kernels. However, the four systems varied vastly at all levels; hence, it is difficult to compare memory models directly. There are several proposals for configurable or hybrid memory systems [Mai et al. 2000; Sankaralingam 2004; Jayasena 2005; Li et al. 2005]. In such systems, a level in the memory hierarchy can be configured as a cache or as a local store depending on an application's needs. Gummaraju and Rosenblum [2005] have shown benefits from a hybrid architecture that uses stream programming on a cache-based superscalar design for scientific code, and later demonstrate a comprehensive framework for dynamic workload scheduling based on stream programming [Gummaraju et al. 2008]. Our work supports this approach, as we show that cache-based memory systems can be as efficient as streaming memory systems, but could benefit in terms of bandwidth consumption and latency tolerance from stream programming.

In some respects, our comparison of streaming and cache-based memory systems echoes previous comparisons of shared memory to message-passing [Klaiber and Levy 1994; Culler et al. 1999]. Shared memory, like traditional caching, couples a conceptually simple programming model with intricate hardware. On the other hand, streaming, like message-passing, couples a pragmatic programming model with conceptually simple hardware.

Although the similarities between streaming and message-passing are compelling for comparison's sake, there is an important difference to acknowledge. Message-passing systems typically depend on a traditional cache-hierarchy for each core's communication with its own secondary memory. Cores in a streaming memory systems rely on bulk transfer mechanisms to communicate with its own secondary memory. Consequently, code for streaming memory systems requires software data management that far exceeds the amount done by message-passing systems. Ironically, some contemporary streaming memory architectures [Gschwind et al. 2005; Ahn et al. 2004] are built with a single, *shared* uniform-access main memory.

There have been several studies discussing the integration of shared memory and message passing in the same system [Heinlein et al. 1994; Kranz et al. 1993]. Philosophically, the integration of prefetching and other streaming memory mechanisms [Gummaraju et al. 2007] into cache-based memory systems follows the same pattern.

9. CONCLUSIONS

The choice of the on-chip memory model has far-reaching implications for CMP systems. In this article, we performed a direct comparison of two competing models: coherent caches and streaming memory.

We conclude that for the majority of applications on CMPs with up to 16 cores, both models perform and scale equally well. For some applications without significant data reuse, streaming has a performance and energy advantage when we scale the number and computational capabilities of the cores. However, the efficiency gap can be bridged by introducing prefetching and nonallocating write policies to cache-coherent systems. We also found some applications for which streaming scales worse than caching due to the redundancies introduced to make the code regular enough for streaming.

Through the adoption of stream programming methodologies, which encourage blocking, macroscopic prefetching, and locality aware task scheduling, cache-based systems are equally as efficient as streaming memory systems. This indicates that there is not a sufficient advantage in building general-purpose cores that follow a pure streaming model, where all local memory structures and all data streams are explicitly managed. We also observed that stream programming is actually easier when targeting cache-based systems rather than streaming memory systems, and that it may be beneficial in scaling coherence and consistency for caches to larger systems.

ACKNOWLEDGMENTS

We sincerely thank Jung Ho Ahn, Bill Mark, Mattan Erez, and Ron Ho for their invaluable comments on early versions of this paper. We are also grateful for the insightful critique of this work by the reviewers. This work was partially funded by DARPA under contract number F29601-03-2-0117, the FCRP Center for Circuit and System Solutions, under contract 2003-CT-888, and NSF under Career Award No. 0546060. Jacob Leverich is supported by an NVIDIA Fellowship and a David Cheriton Stanford Graduate Fellowship.

REFERENCES

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

Agarwal, V., Hrishikesh, M. S., Keckler, S. W., and Burger, D. 2000. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium Computer Architecture*.

Ahn, J. et al. 2004. Evaluating the imagine stream architecture. In *Proceedings of the 31st International Symposium Computer Architecture*.

Andrews, J. and Backer, N. 2005. Xbox360 system architecture. In *Conference Record of Hot Chips 17*. Stanford, CA.

Barroso, L. A. et al. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*.

Chen, Y.-K., Li, E. Q., Zhou, X., and Ge, S. 2006. Implementation of h.264 encoder and decoder on personal computers. *J. Visual Communication and Image Representation 17*, 2, 509–532.

Chiueh, T. 1993. A generational algorithm to multiprocessor cache coherence. In *International Conference on Parallel Processing*. 20–24.

Culler, D., Singh, J. P., and Gupta, A. 1999. *Parallel Computer Architecture: A Hardware/Software Approach*. st. Louis: Morgan Kauffman.

Dally, W. et al. 2003. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 Conference on Supercomputing*.

Davis, J. D., Laudon, J., and Olukotun, K. 2005. Maximizing CMP throughput with mediocre cores. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*.

Drake, M., Hoffmann, H., Rabbah, R., and Amarasinghe, S. 2006. Mpeg-2 decoding in a stream programming language. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium, Rhodes Island (IPDPS)*.

Eatherton, W. 2005. The push of network processing to the top of the pyramid. Keynote presentation at the Symposium on Architectures for Networking and Communication Systems, Princeton, NJ.

Erez, M., Ahn, J. H., Gummaraju, J., Rosenblum, M., and Dally, W. J. 2007. Executing irregular scientific applications on stream architectures. In *Proceedings of the 21st Annual International Conference on Supercomputing*. 93–104.

Fatahalian, K., Knight, T. J., Houston, M. et al. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*.

Foley, T. and Sugerman, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the Graphics Hardware Conference*

Gordon, M. I. et al. 2002. A stream compiler for communication-exposed architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Gschwind, M. et al. 2005. A novel SIMD architecture for the cell heterogeneous chip-multiprocessor. In *Conference Record of Hot Chips 17*.

Gummaraju, J., Coburn, J., Turner, Y., and Rosenblum, M. 2008. Streamware: programming general-purpose multicore processors using streams. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. 297–307.

Gummaraju, J., Erez, M., Coburn, J., Rosenblum, M., and Dally, W. J. 2007. Architectural support for the stream execution model on general-purpose processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. 3–12.

Gummaraju, J. and Rosenblum, M. 2005. Stream programming on general-purpose processors. In *Proceedings of the 38th International Symposium on Microarchitecture*.

Havran, V. 2002. Heuristic ray shooting algorithms. Ph.D. thesis, Czech Technical University in Prague.

HEINLEIN, J., GHARACHORLOO, K., DRESSER, S., AND GUPTA, A. 1994. Integration of message passing and shared memory in the stanford flash multiprocessor. *SIGOPS Oper. Syst. Rev. 28*, 5, 38–50.

HO, R., MAI, K., AND HOROWITZ, M. 2001. The Future of wires. *Proceedings of the IEEE 89*, 4 (Apr.).

HO, R., MAI, K., AND HOROWITZ, M. 2003. Efficient on-chip global interconnects. In *Symposium on VLSI Circuits*. 271–274.

HOROWITZ, M. AND DALLY, W. 2004. How scaling will change processor architecture. In *Proceedings of the International Solid-State Circuits Conference*. 132–133.

INDEPENDENT JPEG GROUP. 1998. IJG's JPEG Software Release 6b.

ITU-T REC. H.264. 2002. ISO/IEC 144496-10 AVC. 2002.

JANI, D., EZER, G., AND KIM, J. 2004. Long words and wide ports: Reinventing the Configurable Processor. In *Proceedings of the Conference Record of Hot Chips 16*. Stanford, CA.

JAYASENA, N. 2005. Memory hierarchy design for steram computing. Ph.D. thesis, Stanford University.

KHAILANY, B., WILLIAMS, T., LIN, J., LONG, E., RYGH, M., TOVEY, D., AND DALLY, W. 2008. A programmable 512 gops stream processor for signal, image, and video processing. *IEEE Journal of Solid-State Circuits 43*, 1, 202–213.

KLAIBER, A. C. AND LEVY, H. M. 1994. A comparison of message passing and shared memory architectures for data parallel programs. In *Proceedings of the 21th International Symposium on Computer Architecture*.

KONGETIRA, P. 2004. A 32-way Multithreaded sparc processor. In *Proceedings of the Conference Record of Hot Chips*.

KRANZ, D., JOHNSON, K., AGARWAL, A., KUBIATOWICZ, J., AND LIM, B.-H. 1993. Integrating message-passing and shared-memory: early experience. In *Proceedings of the 4th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 54–63.

KUMAR, R., ZYUBAN, V., AND TULLSEN, D. M. 2005. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the 32nd International Symposium on Computer Architecture*.

LEVERICH, J., ARAKIDA, H., SOLOMATNIKOV, A., FIROOZSHAHIAN, A., HOROWITZ, M., AND KOZYRAKIS, C. 2007. Comparing memory systems for chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 358–368.

LEWIS, B. AND BERG, D. J. 1998. *Multithreaded Programming with Pthreads*. Upper saddle River. NJ: Prentice Hall.

LI, M. et al. 2005. ALP: efficient support for all levels of parallelism for complex Media applications. Tech. Rep. UIUCDCS-R-2005-2605, UIUC CS. July.

LIM, A. W., LIAO, S.-W., AND LAM, M. S. 2001. Blocking and array contraction across arbitrarily nested loops using affine partitioning. *ACM SIGPLAN Notices 36*, 7, 103–112.

LIN, Y. 2004. A programmable Vector coprocessor architecture for wireless applications. In *Proceedings of the 3rd Workshop on Application Specific Processors*.

LOGHI, M. AND PNCINO, M. 2005. Exploring energy/performance tradeoffs in shared memory MPSoCs: Snoop-based cache coherence vs. software solutions. In *Proceedings of the Design Automation and Test in Europe Conference*

MACHNICKI, E. 2005. Ultra high performance scalable DSP family for multimedia. In *Proceedings of the Conference Record of Hot Chips 17*.

MAI, K. et al. 2000. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th International Symposium on Computer architecture*.

MIPS32 2001. MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set. MIPS Technologies, Inc.

MOSHOVOS, A. 2005. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of the 32nd International Symposium on Computer Architecture*.

MPEG SOFTWARE SIMULATION GROUP. Mssg mpeg2 encoder and decoder. Available at: http://www.mpeg.org/MPEG/MSSG/.

SANKARALINGAM, K. 2004. TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. *ACM Trans. Archit. Code Optim. 1*, 1, 62–93.

SUH, J. et al. 2003. A performance analysis of PIM, stream processing, and tiled processing on memory-intensive signal processing kernels. In *Proceedings of the 30th International Symposium on Computer Architecture*.

Tarjan, D., Thoziyoor, S., and Jouppi, N. P. 2006. CACTI 4.0. Tech. Rep. HPL-2006-86, HP Labs.

Taylor, M. et al. 2004. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the 31st International Symposium on Computer Architecture*.

Tensilica 2007. Tensilica Software Tools. http://www.tensilica.com/products/software.htm.

VanderWiel, S. P. and Lilja, D. J. 2000. Data prefetch mechanisms. *ACM Computing Surveys 32*, 2, 174–199.

Wang, D. et al. 2005. DRAMsim: A memory-system simulator. *SIGARCH Computer Architecture News 33*, 4.

Wang, Z. et al. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*.

Wang, Z. et al. 2003. Guided region prefetching: a cooperative hardware/software approach. In *Proceedings of the 30th International Symposium on Computer Architecture*.

Yeh, T.-Y. 2005. The low-power high-performance architecture of the PWRficient processor family. In *Proceedings of the Conference Record of Hot Chips 17*.

Received March 2007; revised June 2008; accepted July 2008