Cache-Conscious Thread Scheduling for Massively Multithreaded Processors

Highly multithreaded architectures introduce another dimension to fine-grained hardware cache management. The order in which the system's threads issue instructions can significantly impact the access stream seen by the caching system. This article presents a cache-conscious wavefront scheduling (CCWS) hardware mechanism that uses memory system feedback to guide the issue-level thread scheduler and shape the access pattern seen by the first-level cache.

Timothy G. Rogers
University of British Columbia

Mike O'Connor
AMD Research

Tor M. Aamodt
University of British Columbia

The past 30 years have seen significant research and development devoted to using on-chip caches more effectively. At the hardware level, cache management has typically been optimized by improvements to the hierarchy, replacement or insertion policy, coherence protocol, or some combination of these. Previous work on hardware caching assumes that the access stream seen by the memory system is fixed. However, massively multithreaded systems introduce another dimension to the problem. Each cycle, a fine-grained, issue-level thread scheduler must choose which of a core's active threads issues next. This decision has a significant impact on the access stream seen by the cache. In a massively multithreaded system, there could be more than 1,000 threads ready to be scheduled on each cycle. This article explores this observation and uses the thread scheduler to explicitly manage the access stream seen by the memory system to maximize throughput.

This work's primary contribution is a cache-conscious wavefront scheduling (CCWS) system that uses locality information from the memory system to shape future memory accesses through issue-level thread scheduling. As with traditional attempts to optimize cache replacement and insertion policies, CCWS attempts to predict when cache lines will be reused. However, cache way-management policies' decisions are made among a small set of blocks. A thread scheduler effectively chooses which blocks get inserted into the cache from a pool of potential memory accesses that can be much larger than the cache's associativity. Similar to how cache replacement policies effectively *predict* each line's rereference interval,[1] our proposed scheduler attempts to *change* the rereference interval to reduce the number of interfering references between repeated

Published by the IEEE Computer Society

accesses to high-locality data. Similar to how cache tiling techniques performed in software by the programmer or compiler reduce the cache footprint of inner loops by restructuring the code,[2] CCWS reduces the aggregate cache footprint of many threads by dynamically throttling the number of threads sharing the cache in hardware.

The importance of fine-grained thread scheduling on cache management is a concern to any architecture where many hardware threads can share a cache. Some examples include Intel's Knights Corner,[3] Oracle's Sparc T4,[4] IBM's Blue Gene/Q,[5] and massively multithreaded GPUs like those produced by AMD and Nvidia. In this article, based on our MICRO 2012 paper,[6] we study the effect of thread scheduling on GPUs because they represent the most extreme example of a multithreaded architecture. However, the conclusions and techniques outlined in our work are applicable to any of the architectures previously mentioned or any future design where many threads share a cache.

Effect of shaping the access pattern

To illustrate the effect an issue-level thread scheduler can have on the cache system, consider Figures 1 and 2. They present the access pattern created by two different thread schedulers in terms of cache lines touched. GPUs group threads into wavefronts (or warps) for execution, issuing the same static instruction from multiple threads in a single dynamic instruction. This means one memory instruction can generate up to M data cache accesses where M is the number of threads in a wavefront. In this example, we assume that each instruction generates four memory requests, and we're using a fully associative, four-entry cache with a least recently used (LRU) replacement policy. The access stream in Figure 1 will always miss in the cache. However, the stream in Figure 2 will hit 12 times, capturing every redundant access. The GPU's wavefront scheduler creates the access patterns in these two examples. In Figure 1, the scheduler chose to issue instructions from wavefronts in a round-robin fashion without considering the effect of the resulting access stream on cache performance. The scheduler in Figure 2 prioritized accesses from the same wavefront together (indicated by the dashed boxes), creating a stream of cache accesses where locality is captured by our example cache.

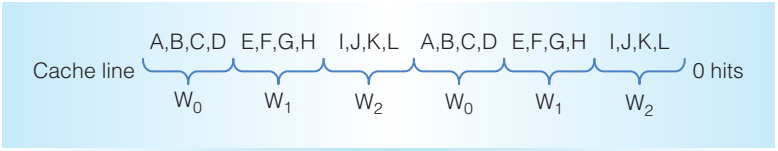

Figure 1. Example access pattern (represented as cache lines touched) resulting from a throughput oriented round-robin scheduler. The letters (A, B, C, D, etc.) represent cache lines accessed. W_i indicates which wavefront generated this set of accesses. For example, the first four accesses to cache lines A, B, C, and D are generated by wavefront 0.

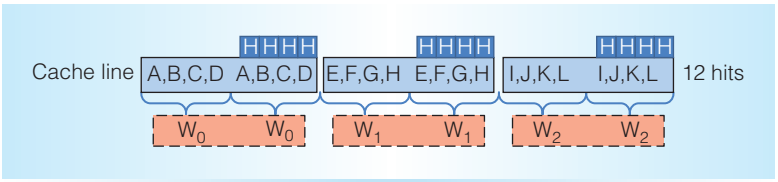

Figure 2. Example access pattern resulting from a scheduler that's aware of the effect scheduling has on the caching system. The dashed boxes indicate that the issue-level scheduler has rearranged the order in which the wavefront's accesses are issued.

GPU architecture

We studied modifications to the GPU-like accelerator architecture illustrated in Figure 3. The workloads were written in OpenCL or CUDA. Initially, an application begins execution on a host CPU, which launches a kernel containing a large number of threads on the GPU. Our baseline system uses GPGPU-Sim 3.x.[7]

Next, we focused on the decision made by the wavefront issue arbiter (WIA) (A in Figure 3). An in-order scoreboard (B) and decode unit (C) control when each instruction in an instruction buffer (D) is ready to issue. The WIA decides which of these ready instructions issues next.

As we mentioned earlier, each memory instruction can generate more than one memory access. Modern GPUs attempt to reduce the number of memory accesses generated from each wavefront using an access coalescer (E), which attempts to coalesce the wavefront's memory requests into cache-line-sized chunks when there is spatial locality

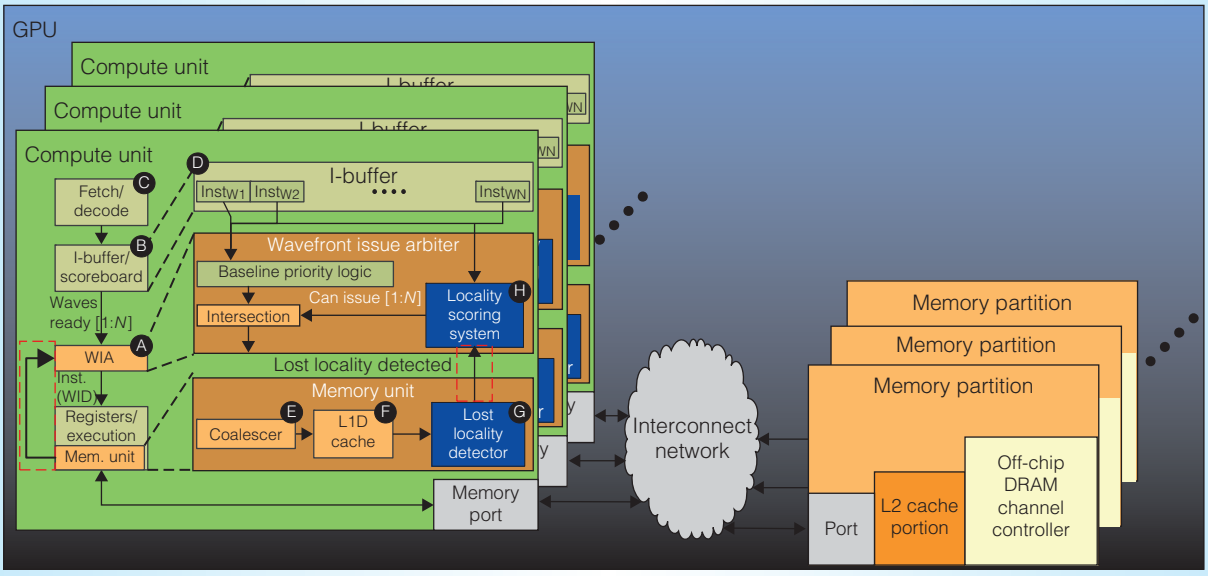

Figure 3. Overview of our GPU-like baseline accelerator architecture. $Inst_{Wi}$ denotes the next instruction ready to issue for wavefront *i*. *N* is the maximum number of wavefront contexts stored on a core. (I-buffer: instruction buffer; L1D cache: level-1 data cache; WID: wavefront ID.)

Table 1. Highly cache-sensitive (HCS) GPU compute benchmarks (CUDA and OpenCL).

Name	Abbreviation
BFS Graph Traversal[8]	BFS
Garbage Collection[9]	GC
K-Means[8]	KMN
Memcached[10]	MEMC

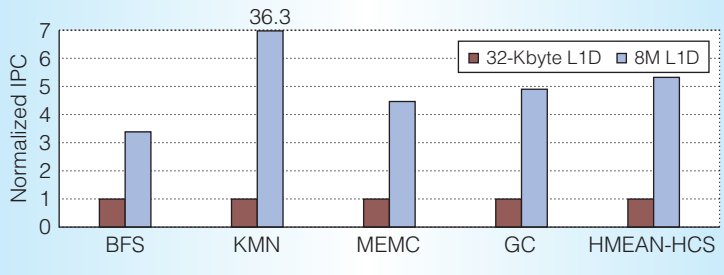

Figure 4. Performance using a round-robin scheduler at various L1 data cache sizes for HCS benchmarks normalized to a cache size of 32 Kbytes. All caches are 8-way set-associative with 128-byte cache lines.

across the threads in a wavefront. Applications with highly regular access patterns could generate as few as two memory requests that service all *M* lanes. Our baseline GPU includes a 32-Kbyte level-1 (L1) data cache (F), which receives memory requests from the coalescer.

L1 data cache performance potential

We focused on improving the performance of a set of economically important server applications, listed in Table 1, running on a GPU. Figure 4 illustrates these benchmarks' cache-size sensitivity when using a round-robin scheduler. All of these applications see a 3× or more performance improvement with a much larger L1 data cache, indicating significant locality within wavefronts executing on the same compute unit.

Locality in cache-sensitive applications

Figure 5 presents the average number of hits and misses per thousand instructions (PKI) of the highly cache-sensitive (HCS) benchmarks we studied using an unbounded L1 data cache. The figure separates hits into two classes. We classify locality that occurs

when data is initially referenced and rereferenced from the same wavefront as intrawavefront locality. Locality resulting from data that is initially referenced by one wavefront and rereferenced by another is classified as interwavefront locality. Figure 5 illustrates that the majority of data reuse observed in our HCS benchmarks comes from intrawavefront locality.

On the basis of this observation, we designed CCWS with the goal of capturing more intrawavefront locality. To this end, CCWS acts as a dynamic thread-throttling mechanism that prevents wavefronts from issuing memory instructions when their accesses are predicted to interfere with intrawavefront locality already present in the cache. CCWS detects if threads require more exclusive cache access through memory system feedback from a lost locality detector (G in Figure 3). The lost locality detector filters victims from the core's L1 data cache by wavefront ID to determine if misses in the L1 data cache have been avoided by giving more exclusive cache access to the wavefront that misses.

CCWS's locality scoring system (H) uses this feedback to decide which wavefront should be permitted to issue memory instructions. Changing the thread scheduler to improve cache effectiveness introduces a new challenge not encountered by traditional cache-management techniques. The scoring system aims at maximizing throughput, not just the cache-hit rate. Typically, massively multithreaded architectures (such as GPUs) hide long latency operations by issuing instructions from more threads. The CCWS scoring system dynamically balances the tradeoff between scheduling fewer threads to improve cache performance and scheduling more threads to improve latency tolerance.

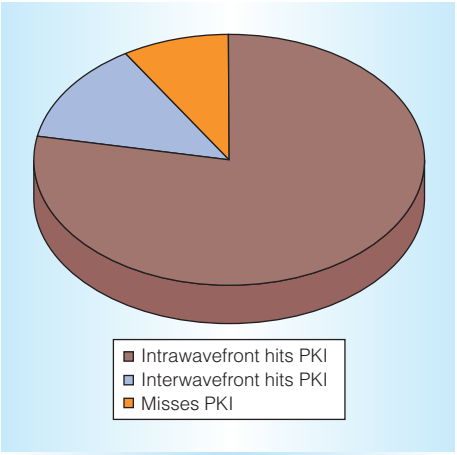

Figure 5. Average hits and misses per thousand instructions (PKI) using an unbounded L1 data cache (with 128-Byte lines) on the HCS benchmarks.

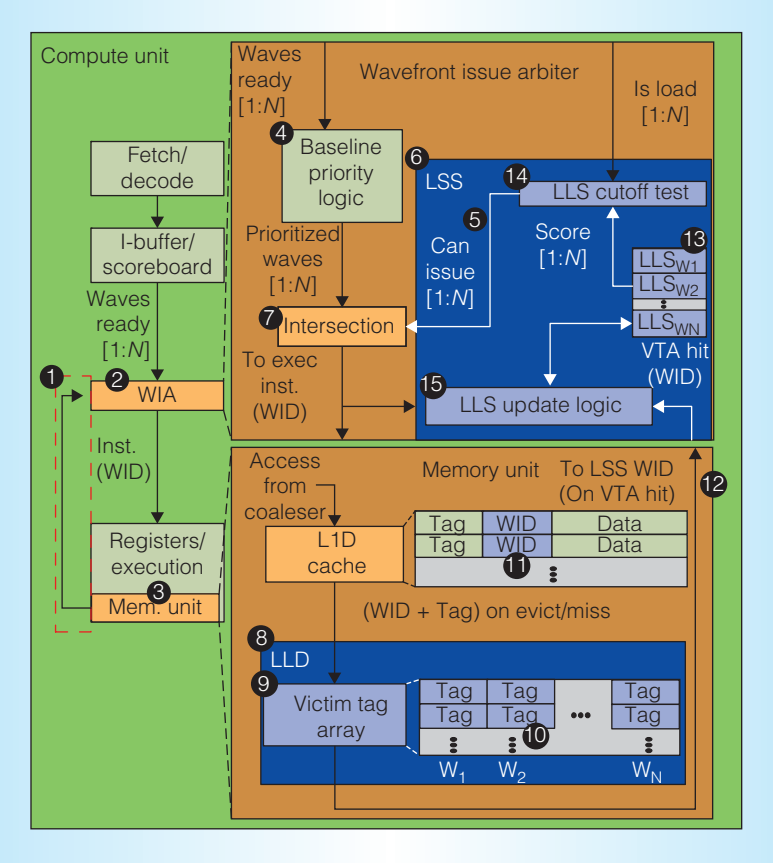

Figure 6. Modeled GPU core microarchitecture. *N* is the number of wavefront contexts stored on a core. (LSS: locality scoring system; LLD: lost intrawavefront locality detector; LLS: lost-locality score; VTA: victim tag array.)

CCWS

CCWS aims to dynamically determine both the number of wavefronts allowed to access the memory system and which wavefronts those should be. Figure 6 presents a more detailed view of the CCWS microarchitecture. At a high level, CCWS is a wavefront scheduler that reacts to access-level feedback (1 in Figure 6) from the L1 data

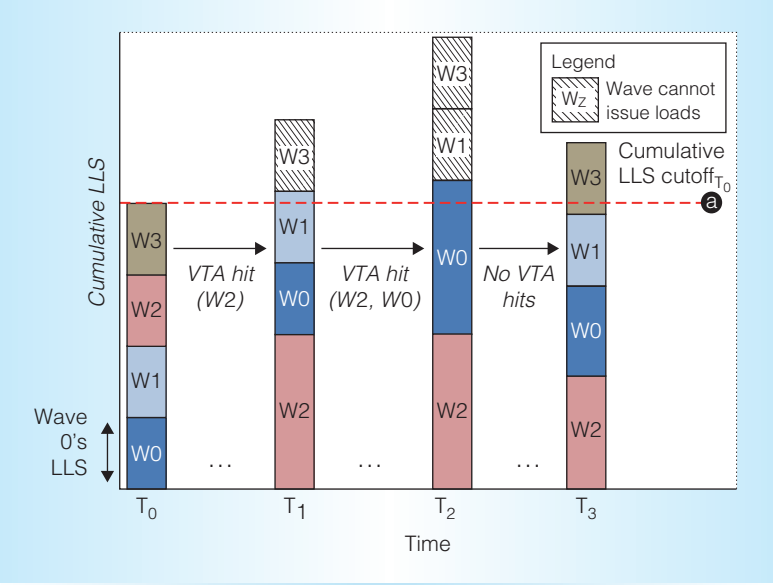

Figure 7. Locality scoring system operation example. Each shaded bar in the graph represents a score given to each wavefront based on how much locality it has lost. The text under each arrow indicates events that have happened in the time interval spanned by the arrow. For example, between T_1 and T_2, wavefronts 2 and 0 have received victim tag array hits, indicating they have lost some intrawavefront locality.

cache and a victim tag array (VTA) at the memory stage.

The locality scoring system makes the CCWS scheduling decisions (6). Figure 7 explains why the scoring system works. At a high level, each wavefront is given a score according to how much intrawavefront locality it has lost. These scores change over time. Wavefronts with the largest scores fall to the bottom of a sorted stack (for example, W_2 at T_1), pushing wavefronts with smaller scores above a cutoff (W_3 at T_1), which prevents them from accessing the L1 data cache. In effect, the locality scoring system reduces the number of accesses between data rereferences from the same wavefront by removing the accesses of other wavefronts.

Effect on baseline issue logic

Figure 6 shows the modifications to the baseline wavefront issue arbiter (2) and memory unit (3) required for CCWS. CCWS is implemented as an extension to the system's baseline wavefront prioritization logic (4). This prioritization could be done in a greedy, round-robin, or two-level manner. CCWS operates by preventing loads that are predicted to interfere with intrawavefront locality from issuing through a "can issue" bit vector (5) output by the locality scoring system (6). The intersection logic block (7) selects the highest priority ready wavefront that has issue permission.

Lost intrawavefront locality detector (LLD)

To evaluate which wavefronts are losing intrawavefront locality, we introduce the lost intrawavefront locality detector (LLD) (8), which uses a VTA (9). The VTA is a highly modified variation of a victim cache.[11] Its sets are subdivided among all the wavefront contexts supported on this core. This gives each wavefront its own small VTA (10). The VTA only stores cache tags and does not store line data. When a miss occurs and a line is reserved in the L1 data cache, the wavefront ID (WID) of the wavefront reserving that line is written in addition to the tag (11). When that line is evicted from the cache, its tag information is written to that wavefront's portion of the VTA. Whenever there is a miss in the L1 data cache, the VTA is probed. If the tag is found in that wavefront's portion of the VTA, the LLD sends a VTA hit signal to the locality scoring system (12). These signals inform the scoring system that a wavefront has missed on a cache line that might have been a hit if that wavefront had more exclusive access to the L1 data cache.

Locality scoring system operation

Figure 7 shows the locality scoring system's operation. In this example, four wavefronts are initially assigned to the core (or compute unit). Time T_0 corresponds to the time these wavefronts are initially assigned to the core. Each segment of the stacked bar represents a score given to each wavefront to quantify the amount of intrawavefront locality it has lost (which is related to the amount of cache space it requires). We call these values the *lost-locality score* (LLS). At T_0 we assign each wavefront a constant base locality score. LLS values are stored in a max heap (13) inside the locality scoring system. A wavefront's LLS can increase when the LLD sends a VTA hit signal for

this wavefront. Each score decreases by one point each cycle until it reaches the base locality score. The locality scoring system gives wavefronts losing the most intra-wavefront locality more exclusive L1 data cache access by preventing the wavefronts with the smallest LLS from issuing load instructions. Wavefronts whose LLS falls above the cumulative LLS cutoff (A in Figure 7) in the sorted heap are prevented from issuing loads.

The LLS cutoff test block (14) takes in a bit vector from the instruction buffer indicating what wavefronts are attempting to issue loads. It also takes in a sorted list of LLS values on a prefix sum, and clears the "can issue" bit for wavefronts attempting to issue loads whose LLS is above the cutoff. In our example from Figure 7, between T_0 and T_1, W_2 has received a VTA hit and its score has been increased. W_2's higher score has pushed W_3 above the cumulative LLS cutoff, clearing W_3's "can issue" bit if it attempts to issue a load instruction. From a microarchitectural perspective, LLS values are modified by the score update logic (15). The update logic block receives VTA hit signals (with a WID) from the LLD, which triggers a change to that wavefront's LLS. In the example, between T_1 and T_2, both W_2 and W_0 have received VTA hits, pushing W_3 and W_1 above the cutoff. Between T_2 and T_3, no VTA hits have occurred and the scores for W_2 and W_0 have decreased enough to allow both W_1 and W_3 to issue loads again. This illustrates how the system naturally backs off thread throttling over time. Our paper explains the scoring system in more detail.[6]

Evaluation

We model CCWS in GPGPU-Sim[7] (version 3.1.0). The simulator is configured to model an Nvidia Quadro FX5800, extended with L1 data caches and a L2 unified cache similar to Nvidia's Fermi. We also evaluate the L1 data cache's hit rate using the Belady-optimal replacement policy,[12] which chooses the line which is rereferenced furthest in the future for eviction. We evaluate Belady replacement using a trace-based cache simulator that takes GPGPU-Sim cache access traces as input. The full version

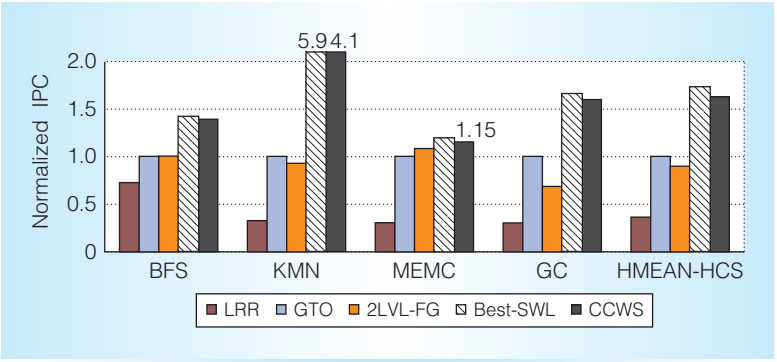

Figure 8. Performance of various schedulers on the HCS benchmarks, normalized to the GTO scheduler.

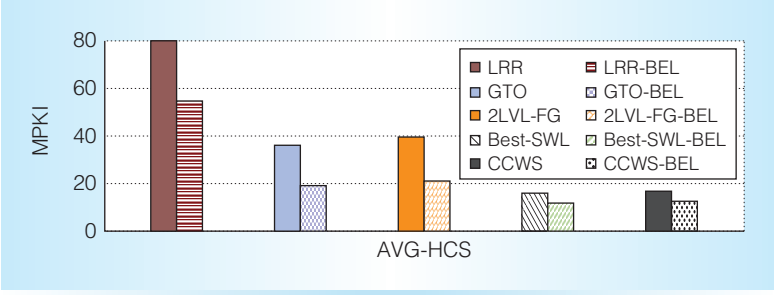

Figure 9. Misses per thousand instructions (MPKI) of various schedulers and replacement policies for the HCS benchmarks.

of our paper provides more details on our experimental setup.[6] We collected the data in Figures 8 and 9 using GPGPU-Sim for the following mechanisms:

- *LRR*: Loose round-robin scheduling. Wavefronts are prioritized for scheduling in round-robin order. However, if a wavefront can't issue during its turn, the next wavefront in round-robin order is given the chance to issue.
- *GTO*: A greedy-then-oldest scheduler. GTO runs a single wavefront until it stalls, then picks the oldest ready wavefront.
- *2LVL-GTO*: A two-level scheduler similar to that described by Narasiman et al.[13] Their scheme subdivides wavefronts waiting to be scheduled on a core into fetch groups (FG) and executes from only one FG until all wavefronts in that group are stalled. Intra- and inter-FG arbitration is done in a GTO manner.

- *Best-SWL:* An Oracle solution that knows the optimal number of wavefronts to schedule concurrently before the kernel begins to execute.
- *CCWS:* Cache-conscious wavefront scheduling with GTO wavefront prioritization logic.

The data for Belady-optimal replacement misses per thousand instructions (MPKI) presented in Figure 9 is generated with our trace-based cache simulator. *(Scheduler)-BEL* is the cache-miss rate reported by our cache simulator when using the Belady-optimal replacement policy. The access streams generated by running GPGPU-Sim with the specified (scheduler) are used.

Figure 8 shows that CCWS achieves a harmonic mean performance improvement of 63 percent over a simple greedy wavefront scheduler and 72 percent over the 2LVL-GTO scheduler on HCS benchmarks. The GTO scheduler performs well because prioritizing older wavefronts allows them to capture intrawavefront locality by giving them more exclusive access to the L1 data cache.

CCWS and SWL provide further benefit over the GTO scheduler because these programs have a number of uncoalesced loads, touching many cache lines in relatively few memory instructions. Therefore, even restricting access to the cache to just the oldest wavefronts still touches too much data to be contained by the L1 data cache.

Figure 9 shows the average MPKI for all of our HCS applications using an LRU replacement policy and an oracle Belady-optimal replacement policy. It shows that the reason for the performance advantage provided by the wavefront-limiting schemes is a sharp decline in the number of L1 data misses. Furthermore, it demonstrates that a poor scheduler choice can make any replacement policy less effective.

Current GPU architectures are excellent at accelerating applications with copious parallelism, whose memory access are regular and statically predicable. Modern CPUs feature deep-cache hierarchies and a relatively large amount of cache available per thread, making them better suited for workloads with irregular locality. However, the limited thread count and available memory bandwidth of CPUs prohibit them from exploiting pervasive parallelism. Each design has problems running the important class of highly parallel irregular applications, and the question of how future architectures can accelerate them is an important one.

Many highly parallel, irregular applications are commercially important and found in modern data centers. The HCS benchmarks in our paper include several such workloads, including Memcached,[10] a parallel garbage collector,[9] a K-Means clustering algorithm,[8] and a breadth-first search graph traversal program.[8] We demonstrate that these applications are highly sensitive to L1 cache capacity when naive thread schedulers are used. Furthermore, we show that a relatively small L1 cache can capture their locality and improve performance, provided an intelligent issue-level, thread-scheduling scheme is used.

Dynamically limiting the number of threads actively scheduled on a massively multithreaded machine can improve performance. Guz et al. first defined a machine's multithreading "performance valley" as the degree of multithreading where there are too many threads for their aggregate working set to fit in the cache and not enough threads to hide all the memory latency through multithreading.[14] Intuitively, CCWS dynamically detects when a workload has entered the machine's performance valley and compensates accordingly by scaling down the number of threads sharing the cache.

This work offers a new perspective on fine-grained memory-system management. We believe integrating the cache system with the issue-level thread scheduler to change the access stream seen by the memory system opens up a new direction of research in hardware cache management. CCWS demonstrates that a relatively simple, dynamically adaptive, feedback-driven scheduling system can vastly improve the performance of an important class of applications. MICRO

Acknowledgments

We thank Yale N. Patt, Aamer Jaleel, Wilson Fung, Hadi Jooybar, Inderpreet Singh, Tayler Hetherington, Ali Bakhoda,

and our anonymous reviewers for their insightful feedback. We also thank Rimon Tadros for his work on the garbage collector benchmark. This research was funded in part by a grant from Advanced Micro Devices. The simulation infrastructure used in this project is available at https://www.ece.ubc.ca/~tgrogers/ccws.html.

References

1. A. Jaleel et al., ''High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP),'' *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA 10), ACM, 2010, pp. 60-71.
2. M.E. Wolf et al., ''A Data Locality Optimizing Algorithm,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 91), ACM, 1991, pp. 30-44.
3. ''The Intel Xeon Phi Coprocessor,'' brief, Intel, 2012.
4. M. Shah et al., ''Sparc T4: A Dynamically Threaded Server-on-a-Chip,'' *IEEE Micro,* vol. 32, no. 2, 2012, pp. 8-19.
5. R. Haring et al., ''The IBM Blue Gene/Q Compute Chip,'' *IEEE Micro,* vol. 32, no. 2, 2012, pp. 48-60.
6. T.G. Rogers, M. O'Connor, and T.M. Aamodt, ''Cache-Conscious Wavefront Scheduling,'' *Proc. 45th IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2012, pp. 72-83.
7. A. Bakhoda et al., ''Analyzing CUDA Workloads Using a Detailed GPU Simulator,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 09), IEEE CS, 2009, pp. 163-174.
8. S. Che et al., ''Rodinia: A Benchmark Suite for Heterogeneous Computing,'' *Proc. IEEE Int'l Symp. Workload Characterization* (IISWC 09), IEEE CS, 2009, pp. 44-54.
9. K. Barabash and E. Petrank, ''Tracing Garbage Collection on Highly Parallel Platforms,'' *Proc. Int'l Symp. Memory Management* (ISMM 10), ACM, 2010, doi:10.1145/1806651.1806653.
10. T.H. Hetherington et al., ''Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems,'' *Proc. Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 12), IEEE CS, 2012, pp. 88-98.
11. N.P. Jouppi, ''Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,'' *Proc. Int'l Symp. Computer Architecture* (ISCA 90), 1990, pp. 364-373.
12. L.A. Belady, ''A Study of Replacement Algorithms for a Virtual-Storage Computer,'' *IBM Systems J.,* vol. 5, no. 2, 1966, pp. 78-101.
13. V. Narasiman et al., ''Improving GPU Performance via Large Warps and Two-Level Warp Scheduling,'' *Proc. Int'l Symp. Microarchitecture* (MICRO 44), ACM, 2011, pp. 308-317.
14. Z. Guz et al., ''Many-Core vs. Many-Thread Machines: Stay Away From the Valley,'' *IEEE Computer Architecture Letters,* Jan. 2009, pp. 25-28.

Timothy G. Rogers is a PhD candidate in the Department of Electrical and Computer Engineering at the University of British Columbia. His research interests include many-core accelerators. Rogers has a BEng in electrical engineering from McGill University.

Mike O'Connor is a Senior Research Scientist at NVIDIA. This work was performed while he was at AMD Research. O'Connor has an MS in electrical engineering from the University of Texas at Austin. He is a senior member of IEEE and a member of the ACM.

Tor M. Aamodt is an associate professor in the Department of Electrical and Computer Engineering at the University of British Columbia. His research interests include many-core accelerators, heterogeneous multicore processors, and analytical performance modeling of processor architectures. Aamodt has a PhD in electrical and computer engineering from the University of Toronto. He is a member of IEEE and the ACM.

Direct questions and comments about this article to Timothy G. Rogers, 2332 Main Mall, Vancouver, BC, Canada V6T 1Z4; tgrogers@ece.ubc.ca.