Cache-Conscious Wavefront Scheduling

Timothy G. Rogers[1] Mike O'Connor[2] Tor M. Aamodt[1]
[1]University of British Columbia [2]Advanced Micro Devices Inc. (AMD)
tgrogers@ece.ubc.ca, mike.oconnor@amd.com, aamodt@ece.ubc.ca

Abstract

This paper studies the effects of hardware thread scheduling on cache management in GPUs. We propose Cache-Conscious Wavefront Scheduling (CCWS), an adaptive hardware mechanism that makes use of a novel intra-wavefront locality detector to capture locality that is lost by other schedulers due to excessive contention for cache capacity. In contrast to improvements in the replacement policy that can better tolerate difficult access patterns, CCWS shapes the access pattern to avoid thrashing the shared L1. We show that CCWS can outperform any replacement scheme by evaluating against the Belady-optimal policy. Our evaluation demonstrates that cache efficiency and preservation of intra-wavefront locality become more important as GPU computing expands beyond use in high performance computing. At an estimated cost of 0.17% *total chip area, CCWS reduces the number of threads actively issued on a core when appropriate. This leads to an average 25% fewer L1 data cache misses which results in a harmonic mean 24% performance improvement over previously proposed scheduling policies across a diverse selection of cache-sensitive workloads.*

1. Introduction

Manycore accelerators, such as GPUs, enable efficient execution of parallel workloads, allowing continued performance improvement with each process node despite diminished voltage scaling [11]. Programming interfaces like OpenCL [24] and CUDA [33] require the user to define the behavior of a single scalar thread which can be run thousands of times across dozens of simple *single instruction multiple data* (SIMD) compute units (also known as shader cores). This type of architecture, sometimes referred to as *single instruction multiple thread* (SIMT) [28], allows the GPU's SIMD core to make progress on multiple threads using a single instruction by grouping them into wavefronts (or warps), thus amortizing the instruction fetch and decode overhead.

Each cycle, a hardware wavefront scheduler must decide which of the multiple active wavefronts execute next. Our work focuses on this decision. The goal of a wavefront scheduler is to ensure the execution pipeline is kept active in the presence of long latency operations. The inclusion of caches on GPUs [32] can reduce the latency of memory operations and act as a bandwidth filter, provided there is some locality in the access stream. Figure 1 presents the average number of hits and misses *per thousand instructions* (PKI) of *highly cache-sensitive* (HCS) and *moderately cache-sensitive* (MCS) benchmark access streams using an unbounded *level one data* (L1D) cache. The figure separates hits into two classes. We classify locality that occurs when data is initially referenced and re-referenced from the same wavefront as *intra-wavefront locality*. Locality resulting from data that is initially referenced by one wavefront and re-referenced by another is classified as *inter-wavefront locality*. Intra-wavefront locality is a combination of intra-thread locality [27] (where data is private to a single scalar thread) and inter-thread locality where data is shared among scalar threads in the same wavefront.

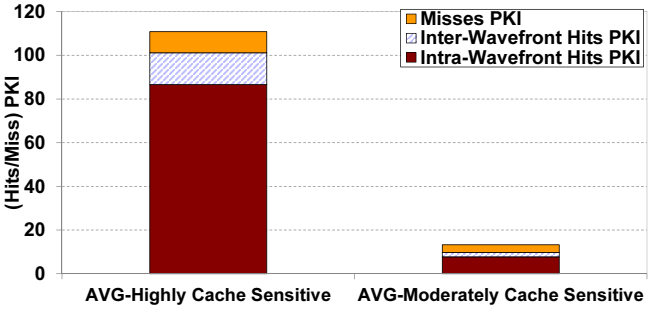


Figure 1: Average hits and misses *per thousand instructions* (PKI) using an unbounded L1 data cache (with 128B lines) on cache-sensitive benchmarks.

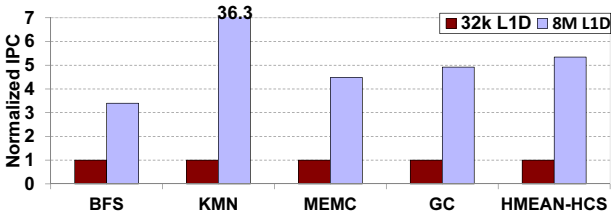
Figure 1 illustrates that the majority of data reuse observed in our HCS benchmarks comes from intra-wavefront locality.

To exploit this type of locality in HCS benchmarks, we introduce Cache-Conscious Wavefront Scheduling (CCWS). CCWS uses a novel *lost intra-wavefront locality detector* (LLD) that alerts the scheduler if its decisions are destroying intra-wavefront locality. Based on this feedback, the scheduler assigns intra-wavefront locality scores to each wavefront and ensures that those wavefronts losing intra-wavefront locality are given more exclusive access to the L1D cache.

Simple wavefront scheduling policies such as round-robin are oblivious to their effect on intra-wavefront locality, potentially touching data from enough threads to cause thrashing in the L1D. A two level scheduler such as that proposed by Narasiman et al. [31] exploits inter-wavefront locality while ensuring wavefronts reach long latency operations at different times by scheduling groups of wavefronts together. However, Figure 1 demonstrates that the HCS benchmarks we studied will benefit more from exploiting intra-wavefront locality than inter-wavefront locality. Existing schedulers do not take into account the effect issuing more wavefronts has on the intra-wavefront locality of those wavefronts that were previously scheduled. In the face of L1D thrashing, the round-robin nature of their techniques will cause the destruction of older wavefront's intra-wavefront locality.

Figure 2 illustrates the cache size sensitivity of our benchmarks (described in Section 4) when using a round-robin scheduler and the baseline system described in Section 4. Although all of these benchmarks are somewhat cache-sensitive, the HCS benchmarks plotted on the left in Figure 2 see 3× or more performance improvement with a much larger L1 data cache.

For GPU-like architectures to effectively address a wider range of workloads, it is critical that their performance on irregular workloads is improved. Recent work on the highly cache-sensitive Memcached (MEMC) [16] and BFS [30] has shown promising results running these commercially relevant irregular parallel workloads on GPUs. However, since current GPUs face many performance challenges running irregular applications, there are relatively few of them written. In this work we evaluate a set of irregular GPU applications and



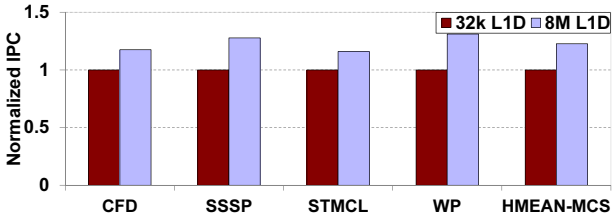


Figure 2: Performance using a loose round-robin scheduler at various L1D cache sizes for highly cache-sensitive (left) and moderately cache-sensitive benchmarks (right), normalized to a cache size of 32k. All caches are 8-way set-associative with 128B cache lines.

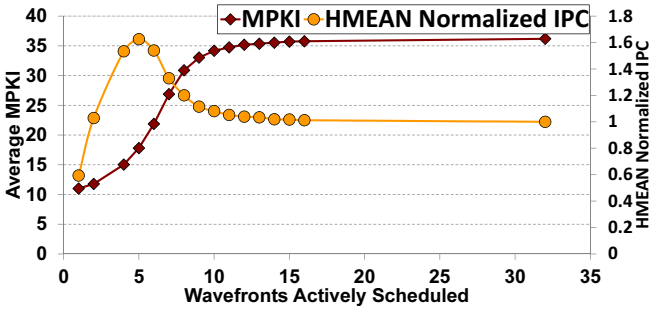


Figure 3: Average *misses per thousand instructions* (MPKI) and *harmonic mean* (HMEAN) performance improvement of HCS benchmarks with different levels of multithreading. *Instructions per cycle* (IPC) is normalized to 32 wavefronts.

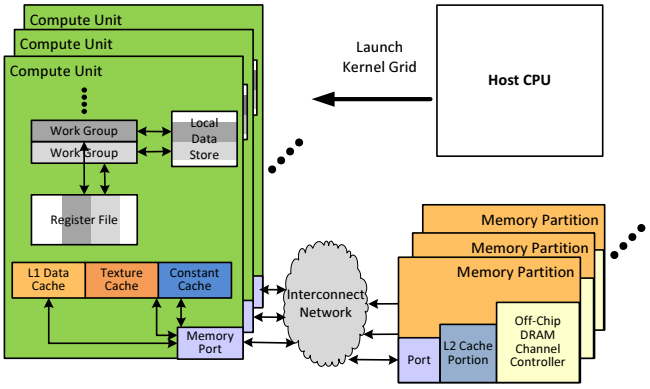


Figure 4: Overview of our GPU-like baseline accelerator architecture.

demonstrate their performance can be highly sensitive to the GPU's wavefront scheduling policy.

Figure 3 highlights the impact wavefront scheduling can have on preserving intra-wavefront locality. It shows the effect of statically limiting the number of wavefronts actively scheduled on a core. Peak throughput occurs at a multithreading value less than maximum concurrency, but greater than the peak cache performance point (which limits concurrency to a single wavefront). Although it may seem counterintuitive to limit the amount of multithreading in a GPU, our data demonstrates a trade-off between hiding long latency operations and creating more of them by destroying intra-wavefront locality.

Our work draws inspiration from cache replacement and insertion policies in that it 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 re-reference interval [21], our proposed scheduler effectively *changes* the re-reference interval to reduce the number of interfering references between repeated accesses to high locality data. Unlike scheduling approaches for managing contention implemented in the operating system [39], our technique exploits fine-grained information available to the low-level hardware scheduler.

This paper makes the following contributions:

- It identifies intra and inter wavefront locality and quantifies the trade-off between maximizing intra-wavefront locality and concurrent multithreading.
- It proposes a novel *Cache-Conscious Wavefront Scheduling* (CCWS) mechanism which can be implemented with no changes to the cache replacement policy. CCWS uses a novel *lost intra-wavefront locality detector* (LLD) to update an adaptive locality scoring system and improves the performance of HCS workloads by 63% over existing wavefront schedulers.
- It demonstrates that CCWS reduces L1D cache misses more than the Belady-optimal replacement scheme.
- It demonstrates that CCWS can be tuned to trade-off power and performance. A power-tuned configuration of CCWS reduces energy-expensive L1D cache misses an additional 18% above the performance tuned configuration while still achieving a 49% increase in performance on HCS workloads.

The rest of this paper is organized as follows, Section 2 describes our baseline architecture, Section 3 describes our scheduling techniques including CCWS, Section 4 describes methodology, Section 5 describes our results, Section 6 describes related work, and Section 7 concludes.

2. Baseline Architecture

In this work we study modifications to the GPU-like accelerator architecture illustrated in Figure 4. The workloads we study are written in OpenCL [24] or CUDA [33]. Initially, an application begins execution on a host CPU and then a kernel is launched on the GPU. An OpenCL kernel is composed of "work items" which can be thought of as scalar threads. To facilitate communication, "work items" are collected into "work groups" which can communicate through local memory. Work items are grouped into wavefronts that execute in lock-step on a GPU core. The microarchitecture of the baseline GPU core with the extensions required to support CCWS is illustrated in Figure 5. The GPGPU-Sim 3.x Manual [1] describes the baseline pipeline in more detail.

Issuing a single wavefront memory instruction can generate up to W data cache accesses where W is the wavefront width. Modern GPUs attempt to reduce the number of memory accesses generated from each wavefront by coalescing the lane's memory requests into cache line sized chunks when there is spatial locality across the wavefront [33]. Applications with highly regular access patterns may generate as few as two memory requests that service all W lanes. Our baseline (L1D) cache evicts global data on writes and reserves cache lines on misses.

Our baseline architecture assigns workgroup sized chunks of threads to compute units. Each compute unit is able to schedule wavefronts from multiple workgroups. The number of workgroups assigned to each core is limited by the total number of threads on the core and the static resource usage of each workgroup [32].

This work focuses on the decision made by the *wavefront issue arbiter* (WIA) (❶ in Figure 5). An in-order scoreboard (❷) and decode unit (❸) control when each instruction in an instruction buffer (I-Buffer ❷) is ready to issue. The WIA decides which of these ready instructions issues next.

3. Wavefront Scheduling to Preserve Locality

This section describes our scheduling techniques, which take advantage of the insights mentioned in Section 1. First, Section 3.1 analyzes wavefront scheduling for locality preservation in an example workload with intra-wavefront locality. Next, Section 3.2 introduces *static wavefront limiting* (SWL) which gives high-level language programmers an interface to tune the level of multithreading. Finally, Section 3.3 describes *Cache-Conscious Wavefront Scheduling* (CCWS), an adaptive hardware scheduler that uses fine-grained memory system feedback to capture intra-wavefront locality.

3.1. A Code Example

Consider the inner loop of a graph processing workload presented in Example 1. The problem has been partitioned by having each scalar thread operate on all the edges of a single node. The adjoining edges of each node are stored sequentially in memory. This type of storage is common in many graph data structures including the highly space efficient compressed sparse rows [6] representation. This workload contains *intra-wavefront locality* resulting from *intra-thread locality* (data's initial reference and subsequent re-references come from the same scalar thread).

The inner loop of each scalar thread strides through attributes of its assigned node's edges sequentially. This sequential access stream has significant spatial locality that can be captured by a GPU's large cache line size (e.g. 128B). If the GPU was limited to just a single thread per compute unit, the memory loads inside the loop would hit in the L1D cache often. In realistic workloads, more than one thousand threads executing this loop will share the same L1D cache.

Example 1 Example graph algorithm kernel run by each scalar thread.

```
int node_degree = nodes[thread_id].degree;
int thread_first_edge = nodes[thread_id].starting_edge;
for ( int i = 0; i < node_degree; i++ ) {
    edge_attribtes = edges[thread_first_edge + i];
    int neighbour_node_id = edge_attributes.node;
    int edge_weight = edge_attributes.weight;
    ...
}
```

We find that if the working set of all the threads is small enough to be captured by the L1D, optimizing both cache efficiency and overall throughput is largely independent of the scheduler choice. In the other extreme, if only one wavefront's working set fits in the cache, optimizing misses would have each wavefront run to completion before starting another. Optimizing performance when the L1D is not large enough to capture all of the locality requires the wavefront scheduler to intelligently trade-off preserving intra-wavefront locality with concurrent multithreading.

If the scheduler had oracle information about the nature of the workload, it could limit the number of wavefronts actively scheduled to maximize performance. This observation motivates the introduction of *static wavefront limiting* (SWL) which allows a high-level programmer to specify a limit on the number of wavefronts actively scheduled per compute unit at kernel launch.

3.2. Static Wavefront Limiting (SWL)

Figure 3 shows the effect limiting the number of wavefronts actively scheduled on a compute unit has on cache performance and system throughput. Current programming API's such as CUDA and OpenCL allow the programmer to specify workgroup size. However, they allow as many wavefronts to run on each compute unit as shared core resources (e.g., registers, shared scratchpad memory) permit. Consequently, even if the programmer specifies small workgroups, multiple workgroups will run on the same compute unit if resources permit. As a result, the number of wavefronts/warps running at once may still be too large a working set for the L1D. For this reason, we propose *static wavefront limiting* (SWL) which is implemented as a minor extension to the wavefront scheduling logic where a register is used to determine how many wavefronts are actively issued, independent of workgroup size.

In SWL, the programmer must specify a limit on the number of wavefronts when launching the kernel. This technique is useful if the user knows the optimal number of wavefronts prior to launching the kernel, which could be determined by profiling.

In benchmarks that make use of work group level synchronization, SWL limits the number of wavefronts running until a barrier, allows the rest of the work-group to reach the barrier, then continues with the same multithreading constraints.

In Section 5 we demonstrate that the optimal number of wavefronts is different for different benchmarks. Moreover, we find this number changes in each benchmark when its input data is changed. This dependence on benchmark and input data makes an adaptive CCWS system desirable.

3.3. Cache-Conscious Wavefront Scheduling (CCWS)

This subsection first defines the goal and high level implementation of CCWS in Section 3.3.1. Next, Section 3.3.2 details how CCWS is applied to the baseline scheduling logic. Section 3.3.3 explains the *lost intra-wavefront locality detector* (LLD), followed by Section 3.3.4 which explains how our locality scoring system makes use of LLD information to determine which wavefronts can issue. Finally, Section 3.3.5 describes the locality score value assigned to a wavefront when lost locality is detected.

3.3.1. High-Level Description The goal of CCWS is to dynamically determine the number of wavefronts allowed to access the memory system and which wavefronts those should be. At a high level, CCWS is a wavefront scheduler that reacts to access level feedback (❹ in Figure 5) from the L1D cache and a *victim tag array* (VTA) at the memory stage. CCWS uses a dynamic locality scoring system to make scheduling decisions.

The intuition behind why our scoring system works can be explained by Figure 6. At a high level, each wavefront is given a score based on how much *intra-wavefront 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 L1D. In effect, the locality scoring system reduces the number of accesses between data re-references from the

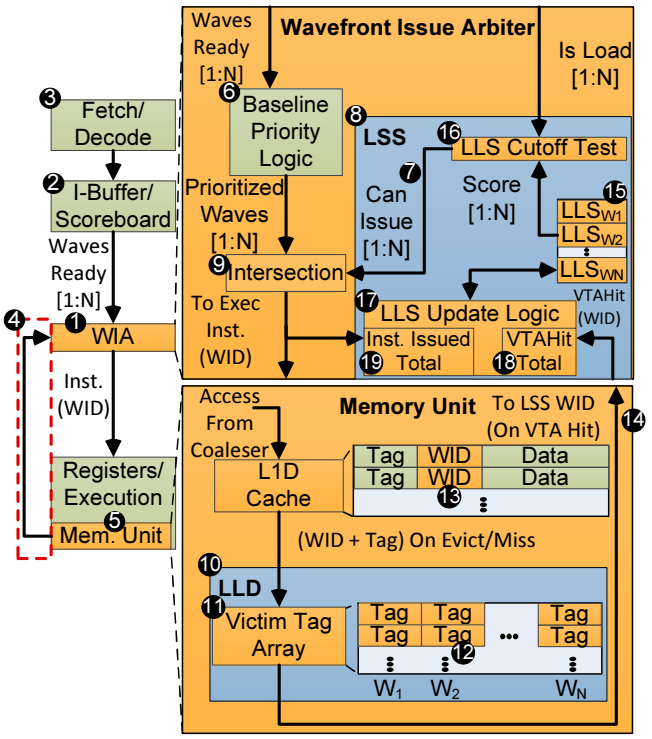


Figure 5: Modeled GPU core microarchitecture. N is the number of wavefront contexts stored on a core. LSS=*locality scoring system*, LLD=*lost intra-wavefront locality detector*, WID=*wavefront ID*, LLS=lost-locality score, VTA=*victim tag array*, I-Buffer=*instruction buffer*

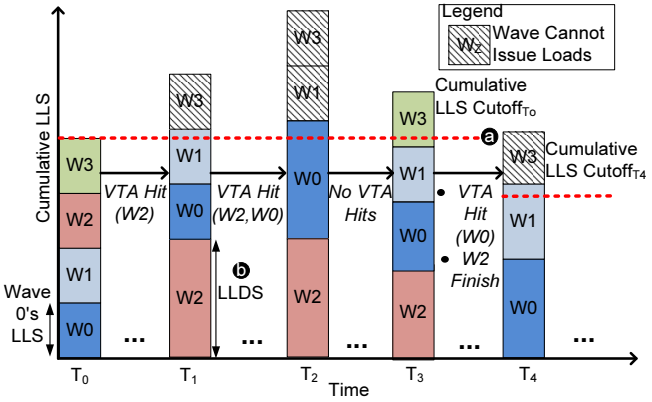


Figure 6: Locality scoring system operation example. LLS=*lost-locality score*, LLDS=*lost-locality detected score*

same wavefront by removing the accesses of other wavefronts. The following subsections describe CCWS in more detail.

3.3.2. Effect on Baseline Issue Logic Figure 5 shows the modifications to the baseline wavefront issue arbiter (❶) and memory unit (❺) required for CCWS. CCWS is implemented as an extension to the system's baseline wavefront prioritization logic (❻). 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 intra-wavefront locality from issuing through a "Can Issue" bit vector (❼) output by the locality scoring system (❽). The intersection logic block (❾) selects the highest priority ready wavefront that has issue permission.

3.3.3. Lost Intra-Wavefront Locality Detector (LLD) To evaluate which wavefronts are losing intra-wavefront locality, we introduce the LLD unit (❿) which uses a *victim tag array* (VTA) (⓫). The VTA is a highly modified variation of a victim cache [23]. The sets of the VTA are sub-divided among the all the wavefront contexts supported on this core. This gives each wavefront its own small VTA (⓬). The VTA only stores cache tags and does not store line data. When a miss occurs and a line is reserved in the L1D cache, the *wavefront ID* (WID) of the wavefront reserving that line is written in addition to the tag (⓭). 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 L1D 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 (⓮). These signals inform the scoring system that a wavefront has missed on a cache line that may have been a hit if that wavefront had more exclusive access to the L1D cache.

3.3.4. Locality Scoring System Operation Figure 6 provides a visual example of the locality scoring system's operation. In this example, there are four wavefronts initially assigned to the compute unit. Time T_0 corresponds to the time these wavefronts are initially assigned to this core. Each segment of the stacked bar represents a score given to each wavefront to quantify the amount of intra-wavefront locality it has lost. We call these values *lost-locality scores* (LLS). At T_0 we assign each wavefront a constant base locality score. LLS values are stored in a max heap (⓯) inside the locality scoring system. A wavefront's LLS can increase when the LLD sends a VTA hit signal for this wavefront. The scores each decrease by one point every cycle until they reach the base locality score. The locality scoring system gives wavefronts losing the most intra-wavefront locality more exclusive L1D cache access by preventing the wavefronts with the smallest LLS from issuing load instructions. Wavefronts whose LLS falls above the cumulative LLS cutoff (ⓐ in Figure 6) in the sorted heap are prevented from issuing loads. The value of the cumulative LLS cutoff is defined as $NumActiveWaves \times BaseLocalityScore$, where $NumActiveWaves$ is the number of wavefronts currently assigned to this core.

The LLS cutoff test block (⓰) 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 LLSs, performs a prefix sum and clears the "Can Issue" bit for wavefronts attempting to issue loads whose LLS is above the cutoff. The locality scoring system is not on the critical path, can be pipelined and does not have to update the score cutoffs every compute unit cycle. In our example from Figure 6, between T_0 and T_1, W_2 has received a VTA hit and its score has been increased to the *lost-locality detected score* (LLDS), (ⓑ in Figure 6). Section 3.3.5 explains the LLDS in more detail. 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 microarchitecture perspective, LLSs are modified by the score update logic (⓱). The update logic block receives VTA hit Signals (with a WID) from the LLD which triggers a change to that wavefront's LLS. We limit the amount one wavefront can dominate the point system by capping each wavefront's score at LLDS, regardless of how many VTA hits it has received. Other methods of capping a wavefront's LLS were attempted and we found that limiting them to the LLDS simplified the point system and yielded the best results. In the example, between T_1 and T_2 both W_2 and W_0 have received VTA hits, pushing both 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. Between time T_3 and T_4, W_2 finishes and W_0 has received a VTA hit to increase its score. This illustrates that when a wavefront is added or removed from the system, the cumulative LLS cutoff changes. Now that there are three wavefronts active, the LLS cutoff becomes $3\times$ the base score. Having the LLS cutoff be a multiple of the number of active wavefronts ensures the locality scoring system maintains its sensitivity to lost-locality. If the LLS cutoff does not decrease when the number of wavefronts assigned to this core decreases, it takes a higher score per wavefront to push lower scores above the cutoff as the kernel ends. This results in the system taking more time to both constrain multithreading when locality is lost and back off thread limiting when there is no lost locality.

3.3.5. Determining the Lost-Locality Detected Score (LLDS) The value assigned to a wavefront's score on a VTA hit (the LLDS) is a function of the total number of VTA hits across all this compute unit's wavefronts (18) and all the instructions this compute unit has issued (19). This value is defined by Equation (1).

$$LLDS = \frac{VTAHits_{Total}}{InstIssued_{Total}} \cdot K_{THROTTLE} \cdot CumLLSCutoff \quad (1)$$

Using the fraction of total VTA hits divided by the number of instructions issued serves as an indication of how much locality is being lost on this core per instruction issued. A constant ($K_{THROTTLE}$) is applied to this fraction to tune how much throttling is applied when locality is lost. A larger constant favors less multithreading by pushing wavefronts above the cutoff value more quickly and for a longer period of time. Finding the optimal value of $K_{THROTTLE}$ is dependent on several factors including the number threads assigned to a core, the L1D cache size, relative memory latencies and locality in the workload. We intend for this constant to be set for a given chip configuration and not require any programmer or OS support. In our study, a single value for $K_{THROTTLE}$ used across all workloads captures 95.4% to 100% of the performance of any workload's optimal $K_{THROTTLE}$ value. This static value is determined experimentally and explored in more detail in Section 5.5. Like the LLS cutoff test, the lost-locality detected score can take several cycles to update and does not impact the critical path.

4. Experimental Methodology

We model the cache-conscious scheduling mechanisms as described in Section 3 in GPGPU-Sim [4] (version 3.1.0) using the configuration in Table 1. The Belady-optimal replacement policy [8], which chooses the line which is re-referenced furthest in the future for eviction, is evaluated using a custom *stand alone GPGPU-Sim cache simulator* (SAGCS). SAGCS is a trace based cache simulator that takes GPGPU-Sim cache access traces as input. Since SAGCS is not a performance simulator and only provides cache information, we do not present IPC results for the Belady-optimal replacement policy. To validate SAGCS, we verified the miss rate for LRU replacement using SAGCS and found that it was within 0.1% of the LRU miss rate reported using GPGPU-Sim. This small difference is a result of variability in the GPGPU-Sim memory system that SAGCS does not take into account.

We perform our evaluation using the high-performance computing GPU-enabled server workloads listed in Table 2 from Rodinia [9], Hetherington et al. [16] and Bakhoda et al. [4]. While

Table 1: GPGPU-Sim Configuration

# Compute Units	30
Wavefront Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L1 Data Cache	32KB, 128B line, 8-way assoc. LRU
L2 Unified Cache	128k/Memory Channel, 128B line, 8-way assoc. LRU
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Memory Controller	out of order (FR-FCFS)
Branch Divergence Method	PDOM [12]
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

Table 2: GPU Compute Benchmarks (CUDA and OpenCL)

Highly Cache Sensitive (HCS)			
Name	**Abbr.**	**Name**	**Abbr.**
BFS Graph Traversal [9]	BFS	Kmeans [9]	KMN
Memcached [16]	MEMC	Garbage Collection [5, 36]	GC
Moderately Cache Sensitive (MCS)			
Name	**Abbr.**	**Name**	**Abbr.**
Weather Prediction [9]	WP	Streamcluster [9]	STMCL
Single Source Shortest Path [4]	SSSP	CFD Solver [9]	CFD
Cache Insensitive (CI)			
Name	**Abbr.**	**Name**	**Abbr.**
Needleman-Wunsch [9]	NDL	Back Propagation [4]	BACKP
Speckle Red. Anisotropic Diff. [9]	SRAD	LU Decomposition [9]	LUD

the regularity of the HPC applications makes them particularly well suited for the GPU, they represent only one segment of the overall computing market [18] [17].

In addition to the cache-sensitive benchmarks introduced earlier, we also evaluate against a number of *cache-insensitive* (CI) benchmarks to ensure CCWS does not have a detrimental effect.

To make use of a larger input, the KMN benchmark was slightly modified to use global memory in place of both texture and constant memory.

All of our benchmarks run from beginning to end which takes between 14 million and 1 billion instructions.

4.1. GPU-enabled server workloads

This work uses two GPU-enabled server workloads. These benchmarks were ported to OpenCL from existing CPU implementations. They represent highly parallel code with irregular memory access patterns whose performance could be improved by running on the GPU.

Memcached-GPU (MEMC) Memcached is a key-value store and retrieval system. Memcached-GPU is described in detail by Hetherington et al. [16]. The application is stimulated with a representative portion of the Wikipedia access trace collected by Urdaneta et al. [37].

Tracing Garbage Collector (GC) Garbage collection is an important aspect of many server applications. Languages such as Java use system-controlled garbage collection to manage resources [2]. A version of the tracing mark-and-compact garbage collector presented in Barabash et al. [5] is created in OpenCL. The collector is stimulated with benchmarks provided by Spoonhower et al. [36].

5. Experimental Results

This section is structured as follows, Section 5.1 presents the performance of SWL, CCWS, other related wavefront schedulers and

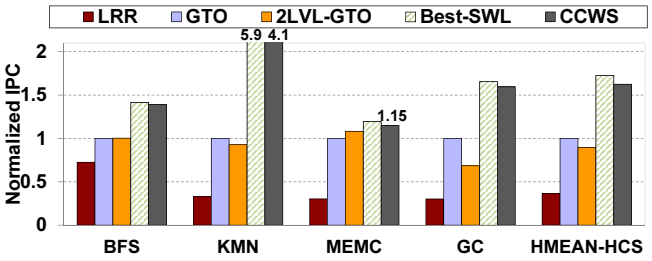


Figure 7: Performance of various schedulers and replacement policies for the highly cache-sensitive benchmarks. Normalized to the GTO scheduler.

the Belady-optimal replacement policy using the system presented in Section 4. The results for CCWS presented in Section 5.1 represent a design point that maximizes performance increase over area increase. The remainder of this section is devoted to exploring the sensitivity of our design and explaining the behaviour of our benchmarks.

5.1. Performance

The data in Figures 7, 8, 9 and 10 is collected 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 cannot 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. The age of a wavefront is determined by the time it is assigned to the core. For wavefronts that are assigned to a core at the same time (i.e. they are in the same workgroup), wavefronts with the smallest threads IDs are prioritized. Other greedy schemes (such as greedy-then-round-robin and oldest-first) were implemented and GTO scheduling had the best results.

2LVL-GTO A two-level scheduler similar to that described by Narasiman et al. [31]. Their scheme subdivides wavefronts waiting to be scheduled on a core into *fetch groups* (FG) and executes from only one fetch group until all wavefronts in that group are stalled. Narasiman et al. used a fetch group size of 8 and a round-robin scheduling policy to select among wavefronts in a fetch group as well as among fetch groups. To provide a fair comparison against their scheduling technique in our simulator and on our workloads, all fetch group sizes were swept. We also explored alternate scheduling policies for intra-FG and inter-FG selection. We found using GTO for both of these policies was better than the algorithm they employed. A fetch group size of 2 using GTO for both intra-FG and inter-FG selection provides the best performance on our workloads and is what we present in our results. This disparity in optimal configuration can be explained by the nature of our workloads and our baseline architecture. Their core pipeline allows only one instruction from a given wavefront to be executing at a time. This means that a wavefront must wait for its previously issued instruction to complete execution before the wavefront can issue another instruction. This is different from our baseline which prevents a fetched instruction from issuing if the scoreboard detects a data hazard.

Best-SWL Static Wavefront Limiting as described in Section 3.2. All possible limitation values (32 to 1) were run and the best performing case is picked. The GTO policy is used to select between wavefronts. The wavefront value used for each benchmark is shown in Table 3.

CCWS Cache-Conscious Wavefront Scheduling described in Section 3.3 with the configuration parameters listed in Table 3. GTO wavefront prioritization logic is used.

The data for Belady-optimal replacement *misses per thousand instructions* (MPKI) presented in Figures 8 and 10 is generated with SAGCS:

<scheduler>-BEL Miss miss rate reported by SAGCS when using the Belady-optimal replacement policy. SAGCS is stimulated with L1D access streams generated by using GPGPU-Sim running the specified <scheduler>. Since SAGCS only reports misses, MPKI is calculated from the GPGPU-Sim instruction count.

Figure 7 shows that CCWS achieves a harmonic mean 63% performance improvement over a simple greedy wavefront scheduler and 72% over the 2LVL-GTO scheduler on HCS benchmarks. The GTO scheduler performs well because prioritizing older wavefronts allows them to capture intra-wavefront locality by giving them more exclusive access to the L1 data cache. The 2LVL-GTO scheduler performs slightly worse than the GTO scheduler because the 2LVL-GTO scheduler will not prioritize the oldest wavefronts every cycle. 2LVL-GTO only attempts to schedule the oldest FG intermittently, once the current FG is completely stalled. This allows loads from younger wavefronts, which would not have been prioritized in the GTO scheduler, to be injected into the access stream, causing older wavefront's data to be evicted.

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 to just the oldest wavefronts still touches too much data to be contained by the L1D. The GTO, 2LVL-GTO, Best-SWL and CCWS schedulers see a greater disparity in the completion time of workgroups running on the same core compared to the LRR scheduler. Since all our workloads are homogeneous (at any given time only workgroups from one kernel launch will be assigned to each core) and involve synchronous kernel launches, the relative completion time of workgroups is not an issue. All that matters is when the whole kernel finishes. Moreover, the highly cache-sensitive workloads we study do not use any workgroup or global synchronization within a kernel launch, therefore older wavefronts are never stalled waiting for younger ones to complete.

Figure 7 also highlights the importance of scheduler choice even among simple schedulers like GTO and LRR. The LRR scheduler suffers from a 64% slowdown compared to GTO. Scheduling wavefronts with a lot of intra-wavefront locality in a RR fashion strides through too much data to be contained in the L1D. Best-SWL is able to slightly outperform CCWS on all the benchmarks. The CCWS configuration used here has been optimized to provide the highest performance per unit area. If the VTA cache is doubled in size, CCWS is able to slightly outperform Best-SWL on some workloads. CCWS is not able to consistently outperform Best-SWL because there is a start-up cost associated with detecting the loss of locality and a cool-down cost to back off the wavefront throttling. Adding to that, the execution time of these kernels is dominated by the code section that benefits from wavefront limiting. Therefore, providing the static scheme with oracle knowledge (through profiling) gives it an advantage over the adaptive CCWS scheme. Section 5.6 examines the shortcomings of the SWL under different run-time condi-

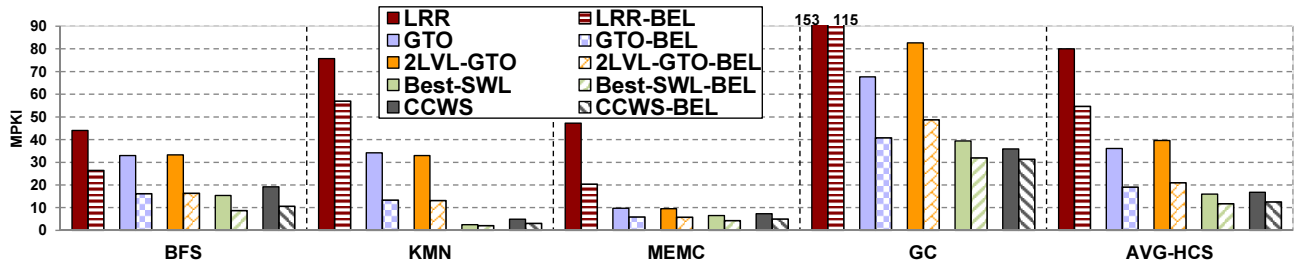


Figure 8: MPKI of various schedulers and replacement policies for the highly cache-sensitive benchmarks.

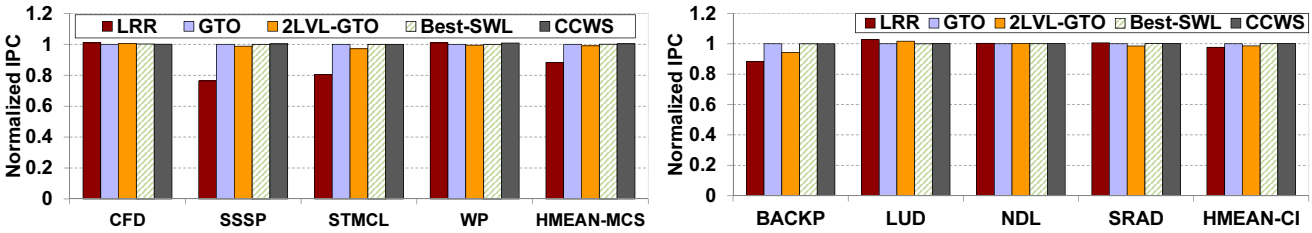


Figure 9: Performance of various schedulers and replacement policies for moderately cache-sensitive (left) and cache-insensitive (right) benchmarks. Normalized to the GTO scheduler.

tions.

Although not plotted here, it is worth mentioning the performance of the 2LVL-LRR scheduling configuration evaluated by Narasiman et al. On the HCS benchmarks the 2LVL-LRR scheduler is a harmonic mean 43% faster than the LRR scheduler, however this is still 47% slower than the GTO scheduler. Performing intra-FG and inter-FG scheduling in a round-robin fashion destroys the intra-wavefront locality of older wavefronts that is captured by the GTO scheduler. However, in comparison to the LRR scheduler, which cycles through 32 wavefronts in a round-robin fashion, cycling through smaller FG sized pools (each fetch group has 8 wavefronts in their configuration) will thrash the L1 data cache less.

Figure 8 illustrates that the reason for the performance advantage of the wavefront limiting schemes is a sharp decline in the number of L1D misses. This figure highlights the fact that no cache replacement policy can make up for a poor choice in wavefront scheduler, as even an oracle Belady-optimal policy on the LRR access stream is outperformed by all the schedulers. The insight here is that even optimal replacement cannot compensate for an access stream that strides through too much data, at least for the relatively low associativity L1 data caches we evaluated.. Furthermore, the miss rate of CCWS outperforms both GTO-BEL and 2LVL-GTO-BEL. This data suggests L1D cache hit rates are more sensitive to wavefront scheduling policy than cache replacement policy.

Figures 9 and 10 present the performance and MPKI of our MCS and CI benchmarks. The harmonic mean performance improvement of CCWS across both the highly and moderately cache-sensitive (HCS and MCS) benchmarks is 24%. In the majority of the MCS and CI workloads, the choice of wavefront scheduler makes little difference and CCWS does not degrade performance. There is no degradation because the MPKI for these benchmarks is much lower than the HCS applications, so there are few VTA hits compared to instructions issued. As a result the lost-locality detected score as defined by Equation (1) stays low and the thread throttling mechanism does not take effect.

5.2. Detailed Breakdown of Inter- and Intra-Wavefront Locality

Figure 11 breaks down L1D accesses into misses, inter-wavefront hits and intra-wavefront hits for all the schedulers evaluated in Section 5.1 on our HCS benchmarks. In addition, it quantifies the portion of intra-wavefront hits that are a result of intra-thread locality. It illustrates that the decrease in cache misses using CCWS and Best-SWL comes chiefly from an increase in intra-wavefront hits. Moreover, the bulk of these hits are a result of intra-thread locality. The exception to this rule is BFS, where 30% of intra-wavefront hits come from inter-thread locality and we see a 23% increase in inter-wavefront hits. An inspection of the code reveals that inter-thread sharing (which manifests itself as both intra-wavefront and inter-wavefront locality) occurs when nodes in the graph share neighbours. Limiting the number of wavefronts actively scheduled increases the hit rate of these accesses because it limits the amount of non-shared data in the cache, increasing the chance that these shared accesses hit.

Table 3: Configurations for Best-SWL (wavefronts actively scheduled) and CCWS variables used for performance data.

Best-SWL		CCWS Config	
Benchmark	Wavefronts Actively Scheduled	Name	Value
BFS	5	$K_{THROTTLE}$	8
KMN	4	Wavefront Base Score	100
MEMC	7	VTA Tag array	8-way
GC	4		16 entries per wavefront
All Others	32		(512 total entries)

Figure 12 explores the access stream of all the cache-sensitive benchmarks using SAGCS and an unbounded L1D. It shows that with the exception of SSSP, the MCS benchmarks have significantly less locality in the access stream. The larger amount of intra-wavefront locality in SSSP is consistent with the significant performance improvement we observe for CCWS at smaller cache sizes when the working set of all the threads does not fit in the L1D cache (see Figure 14).

5.3. Sensitivity to Victim Tag Array Size

Figure 13 shows the effect of varying the VTA size on performance. With a larger victim tag array the system is able to detect lost intra-wavefront locality occurring at further access distances. Increasing the size of the VTA keeps data with intra-wavefront locality in the VTA longer and causes wavefront limiting to be appropriately applied. However, if the VTA size is increased too much, the lost-locality detector's time sensitivity is diminished. The VTA will contain tags from data that was evicted from the L1 data cache so

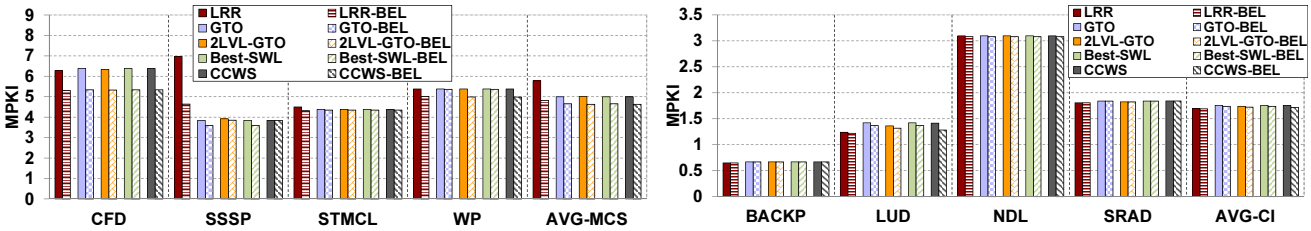


Figure 10: MPKI of various schedulers and replacement policies for moderately cache-sensitive (left) and cache-insensitive benchmarks (right).

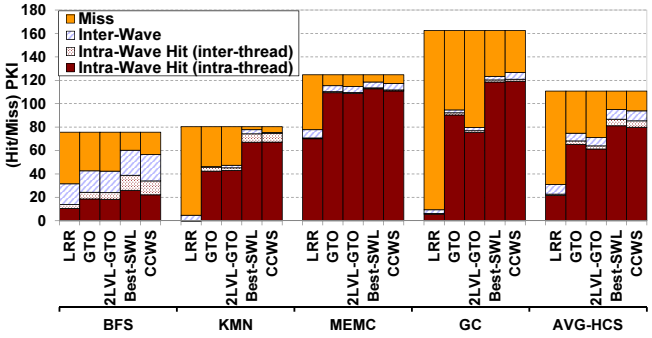


Figure 11: Breakdown of L1D misses, intra-wavefront locality hits (broken into intra-thread and inter-thread) and inter-wavefront locality hits per thousand instructions for highly cache-sensitive benchmarks. The configuration from Section 5.1 is used.

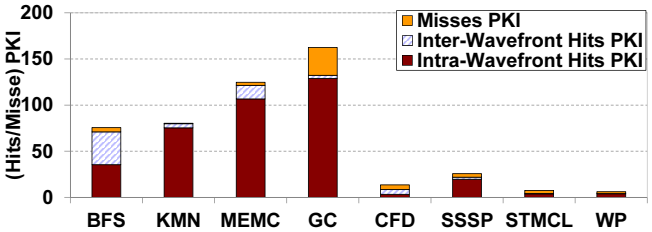


Figure 12: Breakdown of L1D misses, intra-wavefront locality Hits and inter-wavefront locality PKI using an unbounded L1 cache with 128 byte cache lines.

long ago that it would have been difficult to capture with changes to the scheduling policy. For example, at the 512 entry design point, each wavefront has a VTA that can track as much data as the entire L1D. In this configuration, a wavefront would need exclusive access to the L1 data cache to prevent all the detected loss of locality. The increase in detected lost-locality results in excessive wavefront constraining on some workloads. Based on this data, the best-performing configuration with 16 entries per wavefront is selected.

5.4. Sensitivity to Cache Size

Figure 14 shows the sensitivity of CCWS to the L1D size. As the cache size decreases, CCWS has a greater performance improvement relative to the GTO scheduler. This is because at small cache sizes it is even more desirable to limit multithreading to reduce cache footprint. In fact SSSP, which showed no performance gain at 32k shows a 35% speedup when the L1 cache is reduced to 8k. This is because SSSP has significant intra-wavefront locality but its footprint is small enough that it is contained by a 32k L1D. As the cache size increases, the effect of CCWS dwindles relative to the GTO scheduler because the working set of most wavefronts fit in a larger cache. At a large enough cache size, the choice of wavefront scheduler makes little difference.

At 128k per L1D, CCWS shows little benefit over the GTO scheduler. This is because the input to these benchmarks is small enough that 128k captures most of the intra-wavefront locality. Since we are collecting results on a performance simulator that runs several orders of magnitude slower than a real device, the input to our benchmarks is small enough that they finish in a reasonable amount of time. Figure 15 show the effect of increasing the size of the BFS input graph from the baseline 500k edges to 20M edges. As the input size increases, the performance of CCWS over the GTO scheduler also increases even at a 128k L1 cache size. We observe that simply increasing the capacity of the L1 cache only diminishes the performance impact of CCWS with small enough input sets. Hence, we believe CCWS will have an even greater impact on data sizes used in real workloads.

5.5. Sensitivity to $K_{THROTTLE}$ and Tuning for Power

Figure 16 shows the effect of varying $K_{THROTTLE}$ on L1D misses and performance. $K_{THROTTLE}$ is the constant used in Equation (1) to tune the score assigned to wavefronts when lost locality is detected (LLDS). At smaller $K_{THROTTLE}$ values, there is less throttling caused by the point system and more multithreading. At the smallest values of $K_{THROTTLE}$ multithreading is not constrained enough and performance suffers. As $K_{THROTTLE}$ increases, CCWS has a greater effect and the number of L1D misses falls across all the HCS benchmarks. In every HCS benchmark, except GC, performance peaks then falls as $K_{THROTTLE}$ increases. However, since a miss in the L1D cache can incur a significant power cost it may be desirable to use a higher $K_{THROTTLE}$ value to reduce L1D misses at the cost of some performance. For example, at $K_{THROTTLE} = 32$ there is an average 18% reduction in L1D misses over the chosen $K_{THROTTLE} = 8$ design point. $K_{THROTTLE} = 32$ still achieves a 46% performance improvement over the GTO scheduler.

Figure 16 also demonstrates that each benchmark has a different optimal $K_{THROTTLE}$ value. However, the difference in harmonic mean performance between choosing each benchmark's optimal $K_{THROTTLE}$ value and using a constant $K_{THROTTLE} = 8$ is $< 4\%$. For this reason, we do not pursue an online mechanism for determining the value of $K_{THROTTLE}$. If other HCS benchmarks have more variance in their intra-wavefront locality then such a system should be considered.

The value of $K_{THROTTLE}$ makes no difference in the CI benchmarks since there is little locality to lose and few VTA Hits are reported. In the MCS benchmarks there are relatively few L1D MPKI, which keeps the product of $K_{THROTTLE}$ and $\frac{VTAHits_{Total}}{InstIssued_{Total}}$ low. In the MCS benchmarks, CCWS performance matches GTO scheduler performance until $K_{THROTTLE} = 128$. At this point there is a harmonic mean 4% performance degradation due to excessive throttling. Since their performance is largely unchanged by the value of $K_{THROTTLE}$, we do not graph the MCS or CI benchmarks in Figure 16.

5.6. Static Wavefront Limiting Sensitivity

In Section 3 we noted that the optimal SWL limiting number was different for different benchmarks. We also indicated that this value

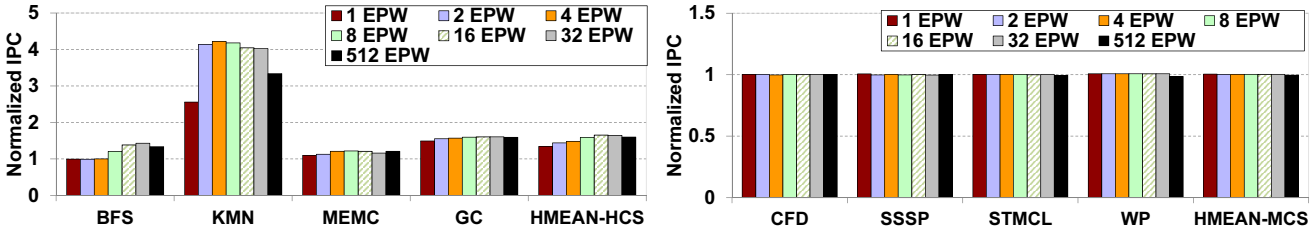


Figure 13: Performance of CCWS at various victim tag array sizes. Normalized to the GTO scheduler. EPW=Entries per Wavefront. EWP 1-4 are 1-4 set associative respectively. All other victim tag arrays are 8-way set associative.

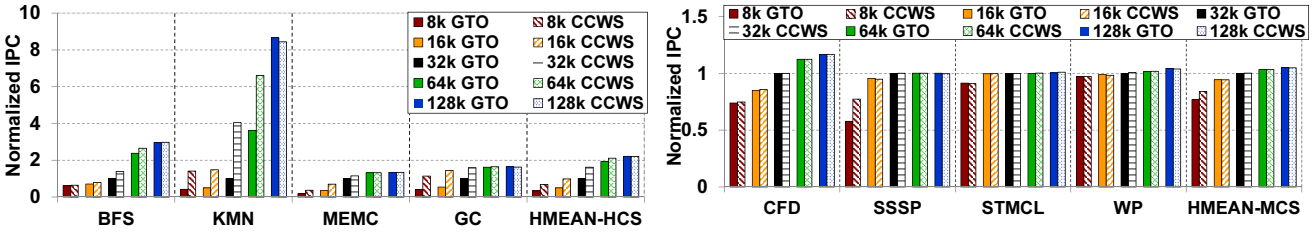


Figure 14: Performance of CCWS and GTO at various cache sizes. Normalized to the GTO scheduler with a 32k L1D. All caches are 8-way set associative. The VTA Size is 16 entries per wavefront for all instances of CCWS.

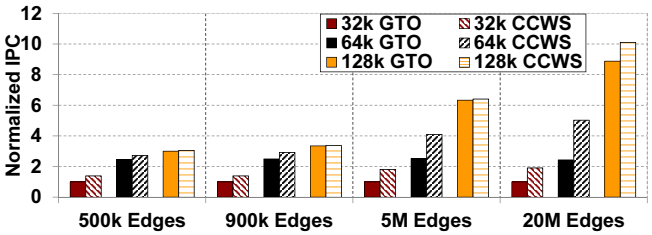


Figure 15: Performance of CCWS on BFS with different graph sizes when varying the L1D cache size and scheduler choice. Normalized to the GTO scheduler with a 32k L1D. The VTA size is 16 entries per wavefront for all instances of CCWS.

changes when running the same benchmark with different input sets. Figure 17 illustrates that peak performance for each of the HCS benchmarks occurs with different multithreading limits. This happens because each workload has a different working set and access stream characteristics. Furthermore, Figure 18 shows that for different input graphs on BFS, the values of the peak performance point are different. This variation happens because the working set size is input data dependent. Finding the optimal wavefront limiting number in SWL would require profiling of each instance of a particular workload, making the adaptive CCWS more practical.

SWL also suffers in programs that have phased execution. The larger and more diverse the application is, the less likely a single wavefront limiting value will capture peak performance. This type of phased behaviour is not abundant in the HCS workloads we study, but as the amount and type of code running on the GPU continues to grow so too will the importance of adaptive multithreading.

SWL is also sub-optimal in a multi-programmed GPU. If wavefronts from more than one type of kernel are assigned to the same compute unit, a per-kernel limiting number makes little sense. Even if there was no cache thrashing in either workload individually their combination may cause it to occur. CCWS will adapt to suit the needs of whatever wavefront combination is running on a compute unit and preserve their intra-wavefront locality. Since there will be no inter-wavefront locality among multi-programmed wavefronts, preservation of intra-wavefront locality becomes even more important.

5.7. Area Estimation

The major source of area overhead to support CCWS comes from the victim tag array. For the configuration used in Table 3 and a 48-bit virtual address space, we require 40 bits for each tag entry in our VTA. Using CACTI 5.3 [38], we estimate that this tag array would consume 0.026 mm^2 per core at 55nm or 0.78 mm^2 for the entire 30 core system. This represents 0.17% of GeForce GTX 285 area, which our system closely models with the exception that we also model data caches. There are a variety of smaller costs associated with our design that are difficult to quantify and as a result are not included in the above estimation. Adding an additional 5-bits to each L1D cache line for the WID costs 160 bytes per core. There are 32 lost-locality score values, each represented in 10 bits which are stored in a max heap. Also, there are two counter registers, one for the number of instructions issued and another for the total VTA hit signals. In addition, there is logic associated with the scoring system. Compared to the other logic in a compute unit, we do not expect this additional logic to be significant.

6. Related Work

This section summarizes and contrasts CCWS against prior scheduling and cache management work.

6.1. Thread Throttling to Improve Performance

Bakhoda et al. [4] present data for several GPU configurations, each with a different maximum number of workgroups (or CTAs) that can be concurrently assigned to a core. They observed that some workloads performed better when less workgroups were scheduled concurrently. The data they present is for a GPU without an L1 data cache, running a round-robin wavefront scheduling algorithm. They conclude that this increase in performance occurs because scheduling less concurrent workgroups on the GPU reduces contention for the interconnection network and DRAM memory system. In contrast, the goal of CCWS is use L1 data cache feedback to preserve locality by focusing on fine-grained, issue level wavefront scheduling, not coarse-grained workgroup assignment.

Guz et al. [15] use an analytical model to quantify the "performance valley" that exists when the number of threads sharing a

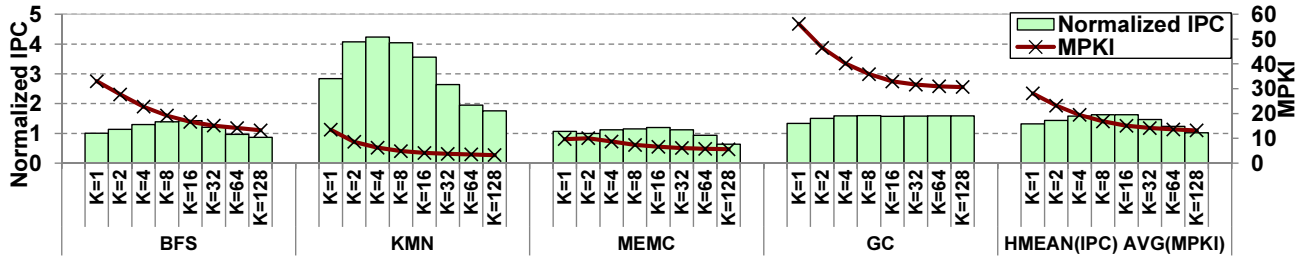


Figure 16: Performance of CCWS (normalized to the GTO scheduler) and MPKI of CCWS when varying $K_{THROTTLE}$.

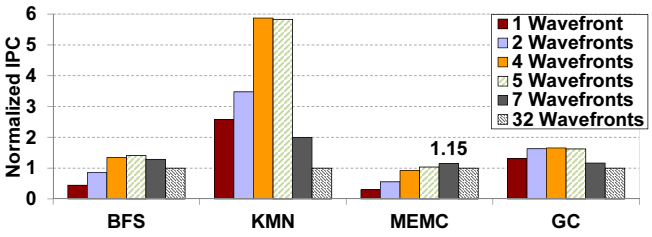


Figure 17: Performance of SWL at various multithreading limits. Normalized to 32 wavefronts.

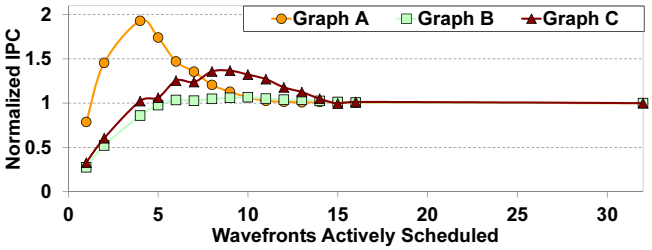


Figure 18: Performance of SWL with different multithreading values on BFS with different input graphs. Normalized to 32 wavefronts.

cache is increased. They show that increasing the thread count increases performance until the aggregate working set no longer fits in cache. Increasing threads beyond this point degrades performance until enough threads are present to hide the system's memory latency. In effect, CCWS dynamically detects when a workload has entered the machine's performance valley and scales down the number of threads sharing the cache to compensate.

Cheng et al. [10] introduce a thread throttling scheme to reduce memory latency in multi-threaded CPU systems. They propose an analytical model and memory task limit throttling mechanism to limit thread interference in the memory stage. Their model relies on a stream programming language which decomposes applications into separate tasks for computation and memory and their technique schedules tasks at this granularity.

6.2. Wavefront Scheduling Techniques

Lakshminarayana and Kim [25] explore numerous warp scheduling policies in the context of a GPU without hardware managed caches and show that, for applications that execute symmetric (balanced) dynamic instruction counts per warp, a fairness based warp and DRAM access scheduling policy improves performance. In contrast to our work, their study did not explore scheduling policies that improve performance by improving cache hit rates.

Fung et al. [12] explore the impact of wavefront scheduling policy on the effectiveness of their *Dynamic Warp Formation* (DWF) technique. DWF attempts to mitigate control flow divergence by dynamically creating new warps when scalar threads in the same wavefront take different paths on a branch instruction. They propose five schedulers and evaluate their effect on DWF. Fung and Aamodt [13] also propose three thread block prioritization schemes to compliment their *Thread Block Compaction* (TBC) technique. The prioritization schemes attempt to schedule threads within the same thread block (or workgroup) together. Their approach is similar to the two-level technique proposed by Narasiman et al. [31], except thread blocks are scheduled together instead of fetch groups. In contrast to both these works, CCWS explores the impact of scheduling on cache locality using existing control flow divergence mitigation techniques.

Gebhart and Johnson et al. [14] introduce the use of a two-level scheduler to improve energy efficiency. Experiments we run using their exact specification yielded mixed results. They note that the performance of their workloads increases less than 10% if a perfect cache is used instead of no cache at all. For this reason, they run all their simulations with a constant 400 cycle latency to global memory. As a result their scheme switches wavefronts out of the active pool whenever a compiler identified global or texture memory dependency is encountered. We find that obeying this constraint causes performance degradation because it does not take cache hits into account. However, if this demotion to the inactive pool is changed to just those operations causing a stall (i.e. those missing in cache) it's operation is similar to Narasiman's two level scheduler we evaluated in Section 5.

Meng et al. [29] introduce *Dynamic Warp Subdivision* (DWS) which splits wavefronts when some lanes hit in cache and some lanes do not. This scheme allows individual scalar threads that hit in cache to make progress even if some of their wavefront peers miss. DWS improves performance by allowing run-ahead threads to initiate their misses earlier and creates a pre-fetching effect for those left behind. DWS attempts to improve intra-wavefront locality by increasing the rate data is loaded into the cache. In contrast, CCWS attempts to load data from less threads at the same time to reduce thrashing.

Narasiman et al. [31] detail a two-level wavefront scheduler similar to that proposed in [14]. Their work focuses on improving performance by allowing groups of threads to reach the same long latency operation at different times. This helps ensure cache and row-buffer locality within a fetch group is maintained and the system is able to hide long latency operations by switching between fetch groups. In contrast, our work focuses on improving performance by adaptively limiting the amount of multithreading the system can maintain based on how much intra-wavefront locality is being lost.

6.3. Improving Cache Efficiency

There is a body of work attempting to increase cache hit rate by improving the replacement policy (e.g., [21] [34] among many others). All these attempt to exploit different heuristics of program behavior to predict a block's re-reference interval and mirror the Belady-optimal [8] policy as closely as possible. While CCWS also at-

tempts to maximize cache efficiency, it does so by shortening the re-reference interval rather than by predicting it. CCWS has to balance the shortening of the re-reference interval by limiting the number of eligible wavefronts while still maintaining sufficient multithreading to cover most of the memory and operation latencies. Other schemes attempt to manage interference among heterogeneous workloads [35, 19] but each thread in our workload has roughly similar characteristics. Recent work has explored the use of prefetching on GPUs [26]. However, prefetching cannot improve performance when an application is bandwidth limited whereas CCWS can help in such cases by reducing off-chip traffic.

Beckmann et al. [7] use victim tag information to detect locality lost due to excessive replication in the cache hierarchy and adapt the replication level accordingly. The LLD in CCWS differs from their technique in that it subdivides the victim tag array by wavefront ID and makes use of the this information to influence thread scheduling.

Concurrent to our work, Jaleel et al. [20] propose the CRUISE scheme which uses LLC utility information to make high level scheduling decisions in multi-programmed CMPs. Our work focuses on the first level cache in a massively multi-threaded environment and is applied at a much finer grain. Scheduling decisions made by CRUISE tie programs to cores, where CCWS makes issue level decisions on which bundle of threads should enter the execution pipeline next.

Agrawal et el. [3] present theoretical cache miss limits when scheduling streaming applications represented as directed graphs on uniprocessors. Their work shows that scheduling the graph by selecting partitions comes within a constant factor of the optimal scheduler when heuristics such as working set and data usage rates are known in advance.

Jia et al. [22] characterize GPU L1 cache locality in a current NVIDIA Tesla GPU and present a compile time algorithm to determine which loads should be cached by the L1D. In contrast to our work, which focuses on locality between different dynamic load instructions, their algorithm and taxonomy focus on locality across different threads in a single static instruction. Moreover, since their analysis is done at compile time they are unable to capture any locality with input data dependence.

7. Conclusion

This work introduces a new classification of locality for GPUs. We quantify the caching and performance effects of both intra- and inter-wavefront locality for workloads in massively multi-threaded environments.

To exploit the observation that intra-wavefront locality is of greatest importance on highly cache-sensitive workloads, this work introduces Cache-Conscious Wavefront Scheduling. CCWS is a novel technique to capitalize on the performance benefit of limiting the number of actively-scheduled wavefronts, thereby limiting L1 data cache thrashing and preserving intra-wavefront locality. Our simulated evaluation shows this technique results in a harmonic mean 63% improvement in throughput on highly cache-sensitive benchmarks, without impacting the performance of cache-insensitive workloads.

We demonstrate that on massively multi-threaded systems, optimizing the low level thread scheduler is of more importance than attempting to improve the cache replacement policy. Furthermore, any work evaluating cache replacement on massively multi-threaded systems should do so in the presence of an intelligent wavefront scheduler.

As more diverse applications are created to exploit irregular parallelism and the number of threads sharing a cache continues to increase on both GPUs and CMPs, so too will the importance of intelligent HW thread scheduling policies, like CCWS.

Acknowledgments

The authors would like to thank Wilson Fung, Hadi Jooybar, Inderpreet Singh, Tayler Hetherington, Ali Bakhoda, the reviewers and our shepherd Yale N. Patt 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 Inc.

References

[1] T. M. Aamodt et al., *GPGPU-Sim 3.x Manual*, http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual, University of British Columbia, 2012.

[2] O. Agesen, D. Detlefs, and J. E. Moss, "Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines," in *Proc. of Prog. Lang. Design and Implementation (PLDI 1998)*, pp. 269–279.

[3] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo, "Cache-Conscious Scheduling of Streaming Applications," in *Proc. of Symp. on Parallelism in Algorithms and Architectures (SPAA 2012)*, pp. 236–245.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. of Int'l Symp. on Performance Analysis of Systems and Software (ISPASS 2009)*, pp. 163–174.

[5] K. Barabash and E. Petrank, "Tracing Garbage Collection on Highly Parallel Platforms," in *Proc. of Int'l Symp. on Memory Management (ISMM 2010)*, pp. 1–10.

[6] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[7] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive Selective Replication for CMP Caches," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 39)*, 2006, pp. 443–454.

[8] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78 –101, 1966.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. of Int'l Symp. on Workload Characterization (IISWC 2009)*, pp. 44–54.

[10] H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang, "Memory Latency Reduction via Thread Throttling," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 43)*, 2010, pp. 53–64.

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2011)*, pp. 365–376.

[12] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 40)*, 2007, pp. 407–420.

[13] W. Fung and T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proc. of Int'l Symp. on High Performance Computer Architecture (HPCA 2011)*, pp. 25 –36.

[14] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2011)*, pp. 235–246.

[15] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. Weiser, "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *Computer Architecture Letters*, vol. 8, no. 1, pp. 25 –28, jan. 2009.

[16] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems," in *Proc. of Int'l Symp. on Performance Analysis of Systems and Software (ISPASS 2012)*, pp. 88 –98.

[17] IDC, "HPC Server Market Declined 11.6% in 2009, Return to Growth Expected in 2010," Mar 2010.

[18] IDC, "Worldwide Server Market Rebounds Sharply in Fourth Quarter as Demand for Blades and x86 Systems Leads the Way," Feb 2010.

[19] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive Insertion Policies for Managing Shared Caches," in *Proc. of Int'l Conf. on Parallel Architecture and Compiler Techniques (PACT 2008)*, pp. 208–219.

[20] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer, "CRUISE: Cache Replacement and Utility-Aware Scheduling," in *Proc. of Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems (ASPLOS 2012)*, pp. 249–260.

[21] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2010)*, pp. 60–71.

[22] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and Improving the use of Demand-Fetched Caches in GPUs," in *Proc. of Int'l Conf. on Supercomputing (ICS 2012)*, pp. 15–24.

[23] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 1990)*, pp. 364–373.

[24] Khronos Group, "OpenCL," http://www.khronos.org/opencl/.

[25] N. B. Lakshminarayana and H. Kim, "Effect of Instruction Fetch and Memory Scheduling on GPU Performance," in *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010.

[26] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 43)*, 2010, pp. 213–224.

[27] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," in *Proc. of Symp. on Principles and Practice of Parallel Programming (PPoPP 2009)*, pp. 101–110.

[28] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, March-April 2008.

[29] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2010)*, pp. 235–246.

[30] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proc. of Symp. on Principles and Practice of Parallel Programming (PPoPP 2012)*, pp. 117–128.

[31] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 44)*, 2011, pp. 308–317.

[32] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf , NVIDIA, 2009.

[33] *NVIDIA CUDA C Programming Guide v4.2*, http://developer.nvidia.com/nvidia-gpu-computing-documentation/, NVIDIA Corp., 2012.

[34] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *Proc. of Int'l Symp. on Computer Architecture (ISCA 2007)*, pp. 381–391.

[35] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proc. of Int'l Symp. on Microarchitecture (MICRO 39)*, 2006, pp. 423–432.

[36] D. Spoonhower, G. Blelloch, and R. Harper, "Using Page Residency to Balance Tradeoffs in Tracing Garbage Collection," in *Proc. of Int'l Conf. on Virtual Execution Environments (VEE 2005)*, 2005, pp. 57–67.

[37] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia Workload Analysis for Decentralized Hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[38] S. Wilton and N. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 5, pp. 677–688, May 1996.

[39] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," in *Proc. of Int'l Conf. on Architecture Support for Prog. Lang. and Operating Systems (ASPLOS 2010)*, pp. 129–142.