View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UT Digital Repository

Copyright
by
Cassidy Aaron Burden
2018

The Report committee for Cassidy Aaron Burden
Certifies that this is the approved version of the following report:

Evaluating Headroom for Smart Caching Policies on GPUs

APPROVED BY

SUPERVISING COMMITTEE:

____
Calvin Lin, Supervisor

____
Akanksha Jain

Evaluating Headroom for Smart Caching Policies on GPUs

by

Cassidy Aaron Burden

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2018

Evaluating Headroom for Smart Caching Policies on GPUs

Cassidy Aaron Burden, M.S.E.
The University of Texas at Austin, 2018

Supervisor: Calvin Lin

This report evaluates two distinct methods of improving the performance of GPU memory systems. Over the past semester, our research has focused on applying a state-of-the-art CPU cache replacement policy on GPUs and exploring headroom of preemptively writing back dirty cache lines.

Our first goal is to reduce L1 and L2 cache miss rates on GPU by implementing the Hawkeye cache replacement policy. Hawkeye calculates the optimal cache replacement policy on previous cache accesses in order to train its predictor for future caching decisions. While some benchmarks show performance improvements with Hawkeye, a significant amount of our benchmarks are not sensitive to the performance of the cache. From our experiments, we show that Hawkeye, on average, gives an IPC improvement of 3.57% and 0.56% over Least Recently Used (LRU) when applied to the L1 and L2 caches respectively.

We also introduce the idea of precleaning, an alternative to write-back or write-through caching that aims to spread out write bandwidth. Committing L2 writes to main memory when memory congestion is low can hide or lower the performance impact of said write. The idea of precleaning shows promise, but evaluating precleaning fully requires more research in GPU access patterns and prediction techniques.

Table of Contents

Abstract **iv**

List of Figures **viii**

Chapter 1. Introduction **1**

1.1 Hawkeye 2

1.2 Precleaning 3

Chapter 2. Background **6**

2.1 GPU Architecture 6

2.2 Cache Eviction and Replacement 7

2.3 Deadblock Predictors 8

Chapter 3. Related Work **9**

3.1 Hawkeye 9

3.2 APCM 11

3.3 Precleaning Related Patents 12

Chapter 4. Solution **13**

4.1 Hawkeye on GPU 13

4.2 Cache Precleaning 15

4.2.1 Final Write Prediction 15

4.2.2 Bandwidth Variation and Prediction 16

Chapter 5. Experiments and Headroom **18**

5.1 Simulator 18

5.2 Cache Sensitivity 20

5.3 Hawkeye 21

5.4 Precleaning 25

Chapter 6. Future Work and Remarks **29**
6.1 Future Work 29
6.2 Remarks and Lessons Learned 30

Chapter 7. Conclusion **32**

Appendices **33**

Appendix A. Hawkeye Experimental Results **34**

Bibliography **41**

List of Figures

1.1 Example of cache cleaning choices, showing potential improvement for bandwidth usage. Traditionally, the two options available were at the time of the original write (i.e. a write through cache) or at time of the eviction (i.e. a write back cache). In this example both write through and write back are arbitrary when trying to optimize for bandwidth usage. We can reason that the better choice is when bandwidth usage is minimal. . . 5

4.1 Cantor pairing function. For the purposes of Hawkeye we use the Cantor pairing function as a rudimentary hashing function between PC and warp ID. 14

4.2 Pseudo-code for common synchronization code pattern in CUDA code. This data is loaded in from DRAM (as this is the only way data can be shared across cores on a GPU) to local shared memory. When computations are done and stored back to global memory, we need to manually synchronize to prevent race conditions in the data. 17

5.1 IPC improvements over baseline when applying Hawkeye to L1 caches. Improvements are measured with both PC and PC+WID features and also with default (4-way associate) and fully associative OPTgen training. FA denotes the fully-associative OPTgen training. 22

5.2 IPC improvements over baseline when applying Hawkeye to L2 caches. 23

5.3 IPC improvement over LRU when write back bandwidth costs are negated. This figure shows us idealized headroom for precleaning. 26

5.4 IPC improvement over LRU when cleaning the least recently used line across all sets. 27

A.1 Cache sensitivity for both L1 and L2 caches. These numbers are percentage improvements over in IPC over baseline when quadrupling the size of each cache. Replacement for all cases is done with LRU. 34

A.2 Average per-feature prediction bias for L1 caches. The bias denotes the percentage of truth values that agree with our predicted value of cache friendly or unfriendly. Here 50% would be equivalent to a random prediction. The final column denotes our average per-feature bias across all benchmarks. 35

A.3 Average per-feature prediction bias for L2 caches. 36

A.4 Average unique values per feature at L1 and L2 caches. 37

A.5 Number of unique PC values seen on CPU SPEC Benchmarks. Notice that these numbers are roughly on the same order of magnitude as the number of PC+WID hashed values we see on GPU. 38

A.6 Average miss rate improvement over LRU for L1 caches. This improvement is measured in percentage points over the miss rate observed with LRU. 39

A.7 Average miss rate improvement over LRU for L2 caches. This improvement is measured in percentage points over the miss rate observed with LRU. 40

Chapter 1

Introduction

GPUs cater to highly parallel and regular data access patterns. While GPUs provide hardware specifically tailored for parallel problems, good GPU performance relies on programs exhibiting behavior that can be partitioned into thousands of highly regular threads executing the same code. Warps are what we call a collection of threads that run in lock-step, and all threads within a warp should run similar code and access relatively nearby data in memory. Data divergence occurs when these warps overload lower-level memory systems by exhibiting irregular access patterns and accessing non-consectutive cache lines. When a program starts to exhibit high levels of data divergence, cache optimizations become more important for retaining good performance. Our goal in this paper, with both Hawkeye and precleaning, is to limit and offset the negative performance impacts of both data divergence and poorly optimized data access patterns.

In this report, we explore two methods for improving performance of GPU memory systems: Hawkeye cache replacement and cache precleaning. We implement Hawkeye, a cache replacement algorithm that learns from a delayed computation of the optimal cache replacement policy [15]. The second

method, which we refer to as precleaning, avoids memory system stalls by committing write-backs at a time of low bandwidth usage rather than at the time of eviction.

1.1 Hawkeye

Computer architects have been using caches and smart cache replacement policies to hide the latency of memory operations for decades [4, 9, 16, 18, 19, 21]. Caches ideally stores all data that will be accessed by the machine in the near future, reducing latency of memory accesses on the critical path. Unfortunately, cache memory is expensive and limited, thus architects rely on cache replacement policies to decide what data will be the most useful in the future.

Least Recently Used (LRU) is the baseline replacement policy for most caches as it is easy to reason about and performs well considering its relative simplicity. LRU does not store any information on lines previously evicted from the cache. Because of this lack of additional meta-data, LRU generally doesn't work well with access patterns with working sets that exceed the size of the cache. A cache with a more sophisticated replacement policy that can handle irregular or complex data access patterns can filter out some of the requests that would otherwise overload the GPU's main memory. By providing a proven approach to cache replacement on CPUs, we hope to improve performance on GPU programs that would previously see no benefit from the cache due to said programs' data access patterns.

Our first idea for improving GPU memory system performance is to use Hawkeye, a CPU cache replacement policy that learns from the theoretically optimal cache replacement decisions for previous accesses to the cache. Hawkeye prefers to evict lines similar to other lower performing lines in the OPTGen algorithm, an online calculation of Belady's optimal cache replacement policy. Hawkeye evaluates similarities between lines by comparing their defining features, such as warp ID and Program Counter (PC). Features help us correlate performance within OPTGen with future performance of lines with identical features. We explore the application of Hawkeye in both the L1 and L2 caches.

1.2 Precleaning

Precleaning, our second idea for improving memory system performance, aims to spread out the bandwidth usage by writing back dirty cache lines at times of lower bandwidth usage. Write-through and write-back caches are the two main ways that modern caches handle mutable data within the cache. Write-through caches immediately write modified data back to their backing store (such as a lower level cache or DRAM). On the other hand, write-back caches delay committing writes to their backing store until the associated line is evicted. This delay in committing writes is possible because non-atomic writes are off the critical path, meaning the processor will never need to stall for a non-atomic write. Write-back caches tend to be more efficient than write-through caches as write-through caches require an access to lower level memory for every single write. While the strategy of writing back at the last possible

moment might be the most convenient, it is not necessarily the best time for data to be written back to lower level memory. Writing back at the time of eviction can be especially problematic in GPUs when unoptimized parallel code causes many simultaneous evictions. The key takeaway is that, rather than a binary choice of write-through and write-back, we actually have a full spectrum of choices to choose from when cleaning dirty cache lines.

For example, a GPU program that needs to load in large chunks of new data at once will incur many evictions in the cache. These evictions will cause write-backs, all clustered around the same time, saturating our bandwidth and stalling the machine. Referring to Figure 1.1 we see that we could make better use of our bandwidth by writing back our dirty lines earlier, rather than at eviction.

We believe this problem is exaggerated on GPUs due to the parallel nature of the hardware. Making accesses to memory across many cores increases the chances of multiple write-backs being triggered around the same time. GPUs rely on having adequate bandwidth available to hide latency across all cores. When bandwidth saturates, code executing on the GPU will effectively start to serialize, which negates the performance benefits of using a GPU.

To further improve GPU memory system performance, we would like to use previous data and current memory bandwidth usage levels to determine a better time to write back modified cache lines. A perfect solution would evenly spread out all write-back traffic, preventing bandwidth spikes that could stall our GPU cores.

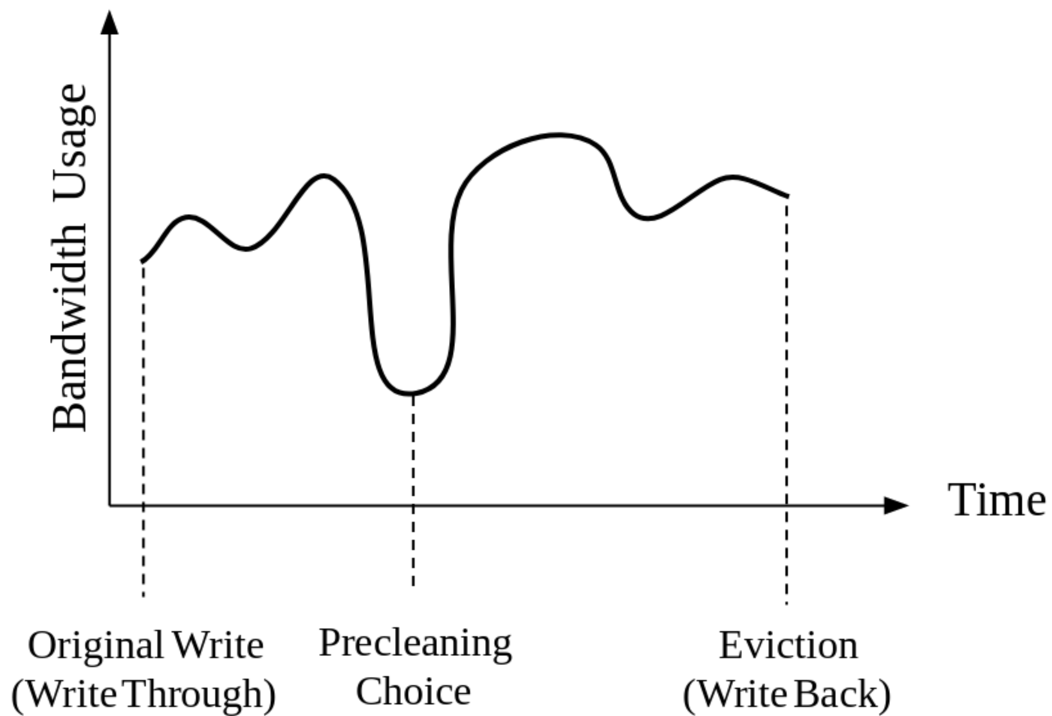

Figure 1.1: Example of cache cleaning choices, showing potential improvement for bandwidth usage. Traditionally, the two options available were at the time of the original write (i.e. a write through cache) or at time of the eviction (i.e. a write back cache). In this example both write through and write back are arbitrary when trying to optimize for bandwidth usage. We can reason that the better choice is when bandwidth usage is minimal.

Chapter 2

Background

2.1 GPU Architecture

GPUs are highly parallel architectures that aim to always have cores running some useful piece of code. GPUs can be thought of as a CPU with dozens of cores and quick context switching. As a core stalls for a memory read or write, another warp (analogous to a lightweight process) is scheduled to that core. If the hardware has enough work to schedule across all cores, maximum throughput is achieved and memory latency can be effectively hidden. This GPU model is highly efficient at parallelizing regular code patterns.

Warps can be broken down even further into threads. Each thread represents a Single Instruction Multiple Data (SIMD) lane. The GPU will have dozens of cores running code in parallel, and each individual core will be running 32 or more instances of the same code as a thread. SIMD greatly increases the amount of parallelization within the hardware, but, as the name suggests, allows a core to only run a single instruction at a time across its threads. Warps rely on their threads to exhibit regular code and data access patterns. Memory stalls and divergent branch paths within threads both negatively impact performance, effectively serializing code execution in the

worst case. Dynamic warp formation [11, 12] and memory coalescing can offset the effects of code and data divergence respectively, but these techniques cannot completely account for the negative effects of poor code design and inefficient memory access patterns.

Each core on the GPU has a register file, an L1 cache, and on chip SRAM referred to as shared memory. All L1 caches are backed by a shared L2 cache, which in turn is backed by DRAM or global memory. Shared memory can be thought of as per-process memory, it is fast and solely owned by the warp that allocates it. Global memory is the only way for different warps to communicate. This memory hierarchy helps to alleviate latency of memory accesses. An access to the L1 or L2 cache is less taxing on warps than waiting on a memory access to DRAM.

2.2 Cache Eviction and Replacement

Cache replacement policies are a highly researched and developed area in computer architecture. Least Recently Used (LRU) is typically used as a baseline metric in papers because it is easy to reason about and gives relatively good performance for its simplicity [9]. However, LRU only learns on basic information currently within the cache (there are no extra data stores for previously evicted lines). Additionally, LRU is susceptible to poor performance given pathological access patterns that can cause thrashing. Early attempts to improve LRU such as Qureshi's et al's DIP paper [21] avoid the effects of thrashing by augmenting LRU to occasionally insert lines into the Most

Recently Used (MRU) position. Over the past couple decades we have seen a trend from heuristic based approaches, such as DIP, to more theoretically grounded replacement policies like EVA [4] and Hawkeye [15].

Belady's OPT algorithm [5] is the optimal cache replacement policy assuming no prefetching, identical cost for all misses, and knowledge of the future. When determining what line to evict, OPT chooses the line that will be reused furthest in the future. Belady's OPT can be used as a ground truth for algorithms attempting to predict optimal cache replacement choices. In the related work section we will briefly elaborate on how Hawkeye calculates OPT online with the OPTGen algorithm, and we will discuss how OPTGen informs Hawkeye's replacement decisions.

2.3 Deadblock Predictors

Deadblock predictors [18] are a natural evolution of basic replacement policies like LRU. Deadblock predictors opt to predict what lines will no longer be reused again in the cache rather than using locally optimal decisions like LRU. Deadblock predictors predict on a variety of features such as timestamps of hits and PC of the hit. We find deadblock predictors interesting for the idea of precleaning because we believe they can be used to also help us predict a write that benefits from precleaning. Predicting a deadblock is similar to predicting when a line will not see writes again in the future. We hope to learn from and alter deadblock predictors like Cache Burst [19] and EVA [4] to predict when a modified line will not be written to again before eviction.

Chapter 3

Related Work

This report draws upon many different concepts in computer architecture cited above, but the three related works we find the most relevant are Hawkeye's original publication by Jain and Lin [15], Koo et al's Access Pattern-Aware Cache Management (APCM) paper [17], and two patents by Marvell and Nvidia that are similar to our proposed idea of precleaning [13, 22].

3.1 Hawkeye

Hawkeye uses OPTGen as an online method for calculating the optimal cache replacement policy. At every cache access, OPTGen tries to match two sequential accesses to a cache line over the program's execution. The time between these sequential accesses represents how long a line needs to be in the cache before getting a hit. For example, a 4-way set associative cache can only have 4 of these cache access intervals overlapping before a line must be evicted or bypassed. OPTGen, like Belady's OPT, always prefers lines that has its next access the earliest. When OPTGen sees a new access, it looks back through the history of all cache accesses to find the last access to this address. By looking back through the history of accesses, OPTGen construct

the exact interval that this line needs to sit in the cache to be a hit. If adding this interval would exceed the associativity of the cache, the line would not be cached by Belady's OPT.

When OPTGen decides to forego caching a line, Hawkeye negatively trains this line's associated feature negatively (cache unfriendly). Additionally, Hawkeye positively trains (cache friendly) features associated with lines that are successfully cached by OPTGen. When Hawkeye needs to evict a cache line it prefers to evict the lines that were trained as cache unfriendly. When all lines are cache friendly, Hawkeye falls back on RRIP to decide which line to evict [16].

OPTGen is implemented in hardware using occupancy vectors. These occupancy vectors keep track of the number of overlapping line intervals after each cache access. When a cache access occurs, OPTGen looks up the last time this line was referenced in the cache. To decide if the interval between these accesses fits in the cache, OPTGen adds 1 to every occupancy vector element that this line's interval spans. If, after incrementing, the occupancy of any element exceeds the associativity of the set, the interval in question is not cached and all occupancy vector changes are rolled back. Unfortunately, with this implementation of occupancy vectors, achieving 100% accurate calculations of Belady's OPT would require infinite memory. The original Hawkeye paper shows that OPTGen can get within 99% accuracy of OPT when using cache sampling and restricting the size of the occupancy vectors.

3.2 APCM

APCM is a paper on GPU cache management improvements that our research draws inspiration from for its cache sensitivity study and its application to L1 caches. APCM provides key insights into the differences between CPU and GPU caches. APCM detects when lines are likely to exhibit reuse between warps of the same core. The cache management policy then pins or bypasses lines to maximize reuse and avoid thrashing within the L1 cache.

To evaluate headroom, APCM measures how sensitive certain GPU benchmarks are to cache performance through its cache sensitivity study. Furthermore, APCM also introduces the idea of sampling warps for its replacement policy meta-data. Rather than adding costly sampling mechanisms to all warps, APCM instead assumes that, because most warps are executing identical code, a single warp's access patterns is representative of all the other warps on the core. This gives us insight into what types of features will be useful for Hawkeye to predict on. Finally, APCM is notable for affirming the idea that more complicated cache replacement policies can be applied at the L1 without significant performance costs. Typically, on CPUs, we would not see cache replacement policies besides LRU replacement at the L1 level due to latency and cost concerns. However, APCM shows that we can benefit from a more complex replacement policy at the L1 level on GPUs.

3.3 Precleaning Related Patents

In our research we found two patents that come close to describing or fully describe our idea of precleaning. The patent by Marvell [22] gives a high level idea of how a precleaning unit works, but unfortunately gives us no insight into the problem due to the lack of details on prediction and precleaning criteria. The patent filed by Nvidia [13] doesn't quite cover our idea of precleaning, but does attempt to optimize away the costs of write backs by using an intermediate cache. From our research we don't believe there is any published prior work that has attempted to implement what we've described as precleaning in this paper. It still remains to be seen if precleaning can provide performance benefits on GPU caches.

Chapter 4

Solution

In this section we cover the specifics of both applying Hawkeye to GPU caching and a cache precleaning system. Our approach for Hawkeye consists of fine tuning and optimizing the predictor for a GPU environment and evaluating the importance of replacement policies on GPUs. Additionally, we present the basic building blocks required for a cache precleaning system such as a modified deadblock predictor and a bandwidth usage prediction scheme.

4.1 Hawkeye on GPU

Adapting Hawkeye to GPUs requires proper training features and adapting caching mechanisms to support said features. We implement Hawkeye with two basic features: the Program Counter (PC) and PC combined with warp ID (PC+WID). Furthermore, we implement and evaluate Hawkeye on both the L1 and L2 cache.

The PC is a rich feature to train on in CPUs and is a common feature in CPU cache replacement policies, prefetchers, and branch predictors. However, kernels on GPUs tend to be much smaller and have fewer accesses to global memory compared to processes on CPUs. These factors lead to a much smaller

number of PCs that access global memory presumably making it harder to use only PC as a feature for Hawkeye. To alleviate this lack of diversity in PC values, we consider warp ID as an additional feature. If we combine PC with the ID of the warp executing the memory access we can achieve a more fine grained set of values to train our predictor on. We use Cantor pairing [20] to act as a hashing mechanism between PC and warp ID. While Cantor pairing gives decent performance for how simple it is, we believe Cantor pairing may be inefficient in hardware because of the multiplication step involved. Refer to Figure 4.1 for a definition of the Cantor pairing function.

$$Cantor(x, y) = \frac{(x + y)(x + y + 1)}{2} + y$$

Figure 4.1: Cantor pairing function. For the purposes of Hawkeye we use the Cantor pairing function as a rudimentary hashing function between PC and warp ID.

In caches we see a trade-off between latency for our access and complexity of our logic. Typically, on CPUs, complex cache replacement policies are reserved for last level caches where high latency is expected and more memory is available for storing meta-data. However, as we've seen in implementations like APCM, L1 caches on GPUs can actually have complex replacement policies while still seeing performance benefits. Furthermore, GPU on-chip memory is shared between the register file, shared memory, and L1 cache and can be used to store meta-data for the L1 cache replacement policies. The additional

meta-data required to implement Hawkeye can come both from the L1 cache itself or additional memory can be repurposed from the register file and shared memory if needed.

4.2 Cache Precleaning

To successfully hide write traffic, a proper solution requires a prediction of: the final write to a line, and the optimal time to commit precleans to lower level memory. Given a trace of the program from previous executions, one could perfectly reorder writes to minimize the impact write backs have on the GPU's memory systems. Our first prediction, determining the final write to a line, considers all lines that are dirty and requires information such as the history of writes to said line and the line's reuse interval. Our second prediction, the optimal time to commit precleans, involves looking at current and past bandwidth usage and using markers in the code and the warp scheduler.

4.2.1 Final Write Prediction

Deadblock predictors are useful for precleaning because they give us an idea of how active a line currently is. It's likely that a deadblock is a good candidate for precleaning, because said line is predicted to have no reads or writes in the future. Though we would like to be more precise than this, we would like a mechanism to predict when a line is a deadblock for the purposes of writes but is still being accessed by reads. Potentially important features

for this predictor include the history of writes to this line, the global history of writes, and which lines will be evicted soon. We believe we can modify a deadblock predictor to predict precleaning candidates by finding when the final write for a line is executed. Currently, this is one of the crucial, unexplored areas for the problem of precleaning. For our experiments we have a basic system using LRU as the indicator for when a line should be precleaned.

4.2.2 Bandwidth Variation and Prediction

A common code pattern in GPU programs is to load data in from global memory into shared memory, do parallel computations, then write back the result of the computations to global memory. Because of the nature of parallel programming, there are often times when different computation units need to synchronize and wait for all other units before continuing. A convenient time to do this synchronization is right after cores write back to global memory. This synchronization ensures that all threads see the latest state of computation before continuing. Please refer to Figure 4.2 for basic pseudo-code that shows this coding pattern.

This synchronization technique yields a useful bandwidth pattern for our technique of precleaning. As cores hit the synchronization barrier their bandwidth usage will drop off, making the synchronization point an opportune time to send out write backs. By getting hints from the compiler or looking at the state of a warp a predictor can infer when threads are attempting to synchronize. We also see much more pronounced dips in bandwidth usage

```
for (int i = 0; i < N; i++) {
  int new_data_idx = getNextPieceOfData(threadIdx);
  int* data = global_data[new_data_idx];
  doComputations(data);
  global_data[new_data_idx] = *data;
  syncThreads();
}
```

Figure 4.2: Pseudo-code for common synchronization code pattern in CUDA code. This data is loaded in from DRAM (as this is the only way data can be shared across cores on a GPU) to local shared memory. When computations are done and stored back to global memory, we need to manually synchronize to prevent race conditions in the data.

between executions of kernels, though kernel termination happens much less often compared to thread synchronization. Again, bandwidth usage prediction still requires more research, for our experiments we usage a basic heuristic based on the number of outstanding memory accesses to DRAM.

Chapter 5

Experiments and Headroom

For our experiments we used GPGPUsim with a variety of programs from the Rodinia 2.0 [8] and Lonestar 2.0 [7] benchmark suites. We chose these programs to see a spread in both regular and irregular data access patterns across our experiments. Later in this section, we present promising results for applying Hawkeye to our GPU simulator environment. However, due to the lack of both time and domain knowledge, we were only able to gather basic headroom data for our idea of cache precleaning. In the following sections we will describe the details of the simulator used, discuss properties of our benchmarks and platform such as cache sensitivity and bandwidth usage, and finally present headroom and actual performance improvements using Hawkeye and precleaning.

5.1 Simulator

We use the GPGPUsim [3] simulator for our experiments, targeting the provided configuration that approximates the Nvidia Fermi architecture. This architecture supports 15 cores, each core consisting of 32 SIMD lanes and a maximum of 48 scheduled warps. Each core has a private 16 KB, 4-

way set associative L1 cache with 128 bytes lines, and the entire system has a shared 786 KB, 8-way set associative L2 cache with 128 byte lines. The L2 cache is split into 12 portions and assigned to the 12 separate DRAM memory partitions. Finally, the L1 cache acts as write-through cache while the L2 cache acts as a write-back cache.

GPGPUsim runs CPU code natively, then intercepts CUDA library calls in order to perform full functional simulation (as opposed to a trace-based simulation). Because GPGPUsim doesn't provide an option for trace-based simulation, we cannot skip warmup cycles and simulations take significant amount of time to run to completion. Due to these restrictions, simulations were sometimes cut short and compared against a baseline that ran for the same amount of time. Unfortunately, this means that our results are often extrapolated and could be inaccurate.

The version of GPGPUsim we use is a slight modification of the 3.0 release that has basic support for CUDA 5.5. This modified version is used in order to be able to run the Lonestar 2.0 benchmark suite. Furthermore, we made use of the AerialVision [2] visualization software provided with GPGPUsim, but we made minor modifications in order to analyze bandwidth patterns while researching precleaning.

The baseline caches within the simulator are all managed by a standard implementation of LRU (no pseudo timestamps are used). Furthermore, as is common on GPUs, the caches have no prefetching and they only cache global reads/writes (accesses to shared memory are on chip and are thus equivalent

to an L1 access). All atomic operations skip the L1 cache, and writes cause evictions from the L1 cache.

5.2 Cache Sensitivity

Cache sensitivity tells us how much cache performance affects a given benchmark. We would expect that a benchmark with heavy code divergence or a small memory footprint would see little to no improvements even from an optimal cache implementation. By enlarging the cache and measuring the change in instructions per clock (IPC) over the baseline, we can get a rough approximation of how much a given benchmark relies on the cache. If we see little to no improvement when enlarging the cache, we wouldn't expect a change in replacement policy to have much affect on performance either. Benchmarks with large IPC improvements have high cache sensitivity and benchmarks that do roughly the same as baseline are considered to have low cache sensitivity. In our Hawkeye experiments, we would like to see large performance improvements on benchmarks that have high cache sensitivity, but see performance on par with LRU in benchmarks with low cache sensitivity. This experiment is motivated by the same study done by the APCM paper as referenced in our Related Work section earlier.

To measure L1 cache sensitivity, we quadruple the size of the L1 cache and compare IPC performance against our baseline cache described above. We also repeat this experiment for the L2 cache. Our results for this experiment are presented in Figure A.1 in Appendix A. Interestingly enough, we see that

a number of graph algorithms including MST, SSSP, and BFS exhibit high levels of cache sensitivity. Graph algorithms on GPU tend to do poorly due to their irregular and hard to predict memory access patterns. The other interesting thing to note here is that between two runs of bfs we see different levels of cache sensitivity, this could be due to how the two data sets were generated or the differing sizes of the data sets. We also see a performance drop in some benchmarks when enlarging the size of the cache. We believe this strange result can be explained by Belady's anomaly [6].

5.3 Hawkeye

For Hawkeye on GPU, we first evaluate both PC and PC combined with warp ID as training features by comparing prediction biases. Next, we evaluate real performance impact by measuring the change in miss rate and IPC over the LRU baseline. Overall, we find that Hawkeye does well compared to LRU, and that, despite our initial thoughts, PC provides better performance than the PC+WID feature at the L1 cache level. Though the best feature to use at the L2 level isn't as obvious a choice.

To begin, we measure the per-feature bias of the OPTGen algorithm for both the PC and PC+WID features. Per-feature bias essentially tells us the prediction quality of our chosen feature, it tells us how often predictions on a feature agree with each other. We note that the HOTSPOT benchmark has a 100% per-feature bias. We believe this high bias is caused by a well optimized code base that provides a small amount of PC values (a max of 4 in

our simulations). The full set of per-feature biases for L1 and L2 caches can be seen in Figures A.2 and A.3 respectively.

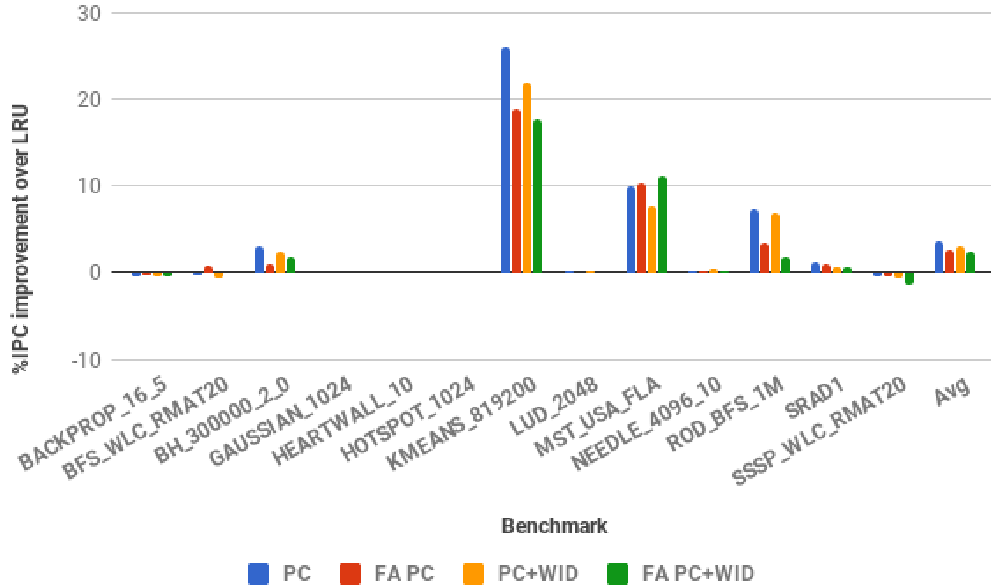

Figure 5.1: IPC improvements over baseline when applying Hawkeye to L1 caches. Improvements are measured with both PC and PC+WID features and also with default (4-way associate) and fully associative OPTgen training. FA denotes the fully-associative OPTgen training.

An important difference between the PC and PC+WID features that's not displayed in our bias metric is the larger number of values being trained on for PC+WID. In Figure A.4 is a plot of the number of unique values seen per feature. This large increase in potential feature values requires adequate hardware to track, and more values being trained typically leads to slower training times. In Figure A.5 we present the number of unique PC values

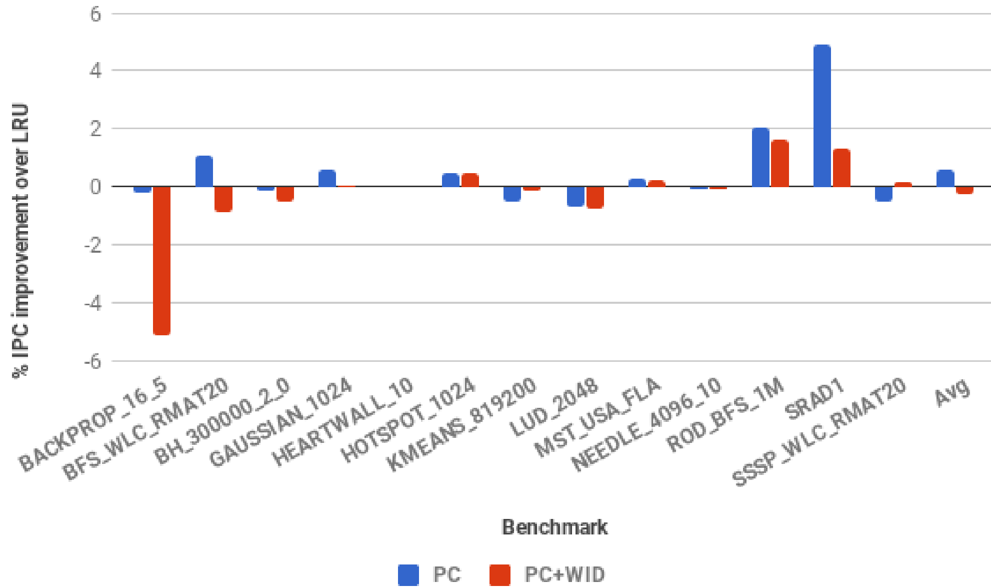

Figure 5.2: IPC improvements over baseline when applying Hawkeye to L2 caches.

seen by OPTGen when running on a last level cache on a CPU using SPEC 2006 [14] benchmarks and ChampSim [1]. We note that the number of unique values seen for PC+WID on GPU and PC on CPU are both on the same order of magnitude. We believe there is more room for complex feature experimentation, especially involving dynamic warp IDs or better hashing algorithms.

In our full performance tests we measure two important metrics: post-warmup cache miss rate, and IPC improvement over baseline. For the post-warmup miss rate metric, we ignore all cache misses before a specified warmup

time, as most misses at the beginning of a program are compulsory and cannot be avoided. We find that a post-warmup time of 3,000,000 cycles fits most of the benchmarks being used (with the exception of HOTSPOT which ends before 3,000,000 cycles). We present the miss rate improvements over LRU for L1 and L2 caches in Figures A.6 and A.7. As mentioned above, PC performs better than the PC+WID feature for all L1 caching benchmarks. We present IPC improvements for L1 and L2 caches in Figures 5.1 and 5.2. IPC improvements appear to roughly correlate with our sensitivity metric from before, for instance KMEANS does the best in both the sensitivity study and in our L1 IPC measurements. However, LUD, which exhibited high sensitivity saw almost no difference from applying Hawkeye at L1.

The performance gains seen at the L1 level are encouraging, as we see significant performance improvements on a few benchmarks. Hawkeye applied to L2 caches gives less significant results, but this is mostly in line with what we found in our cache sensitivity study above. It appears that PC is currently the best feature to use with Hawkeye on GPUs at both the L1 and L2 level. Furthermore, we believe that training OPTgen on a fully-associative cache isn't a good choice as we consistently see performance drops compared to the default OPTGen.

Overall, we see that naively applying Hawkeye to L1 caches gives decent performance improvements. While it is unclear if the cost of Hawkeye at the L1 level can be justified, we believe a simplified, fine-tuned implementation of Hawkeye can give significant performance improvements for a subset of

GPU problems. However, Hawkeye applied at the L2 cache currently gives inconclusive results. On average we see little to no performance improvements and even large dips in performance depending on the benchmark. From these results, we believe the focus of more complex cache replacement should be put towards the L1 cache rather than the L2 cache.

5.4 Precleaning

The results for our idea of precleaning include a coarse measure of headroom and performance tests of the heuristic based predictor we tried. The headroom tests give a idealized view of how much room we have to improve. Like our Hawkeye results, we measure performance benefits in IPC improvement over baseline with no precleaning enabled.

For our headroom study we measure performance improvement when assuming no cost for executing a write-back. In an ideal scenario, a precleaner would be able to completely hide the effects of write-back being sent to main memory. This experiment also gives us an idea of which benchmarks are bandwidth sensitive, much like our cache sensitivity study earlier. We refer to Figure 5.3 for the results of the headroom study. Rodinia's BFS sees a large increase in performance due to the extra bandwidth available. Rodinia's BFS is a clear outlier from the other benchmarks, we hypothesize that this is caused by a underutilization of shared memory causing all writes to go through global memory. If our hypothesis is correct, this would lead us to believe that precleaning has the most potential improvements on unoptimized, ported

code that does not properly make use of CUDA specific concepts like shared memory.

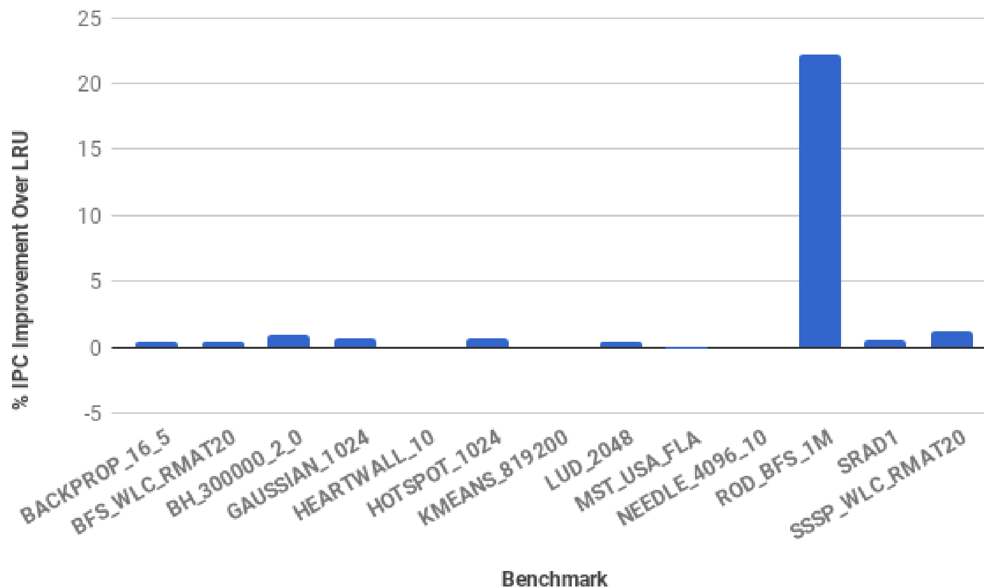

Figure 5.3: IPC improvement over LRU when write back bandwidth costs are negated. This figure shows us idealized headroom for precleaning.

We also evaluate our basic implementation of a precleaning unit. Referring to Figure 5.4 we see little impact to performance with our implementation of a precleaning unit. While we see a small increase in performance from BH, we see a more significant decrease in performance from Rodinia's implementation of BFS. These poor results highlight the difficulty of fine tuning a predictor for precleaning. Much like prefetching, it is easy to make a precleaner too aggressive and actually hurt performance.

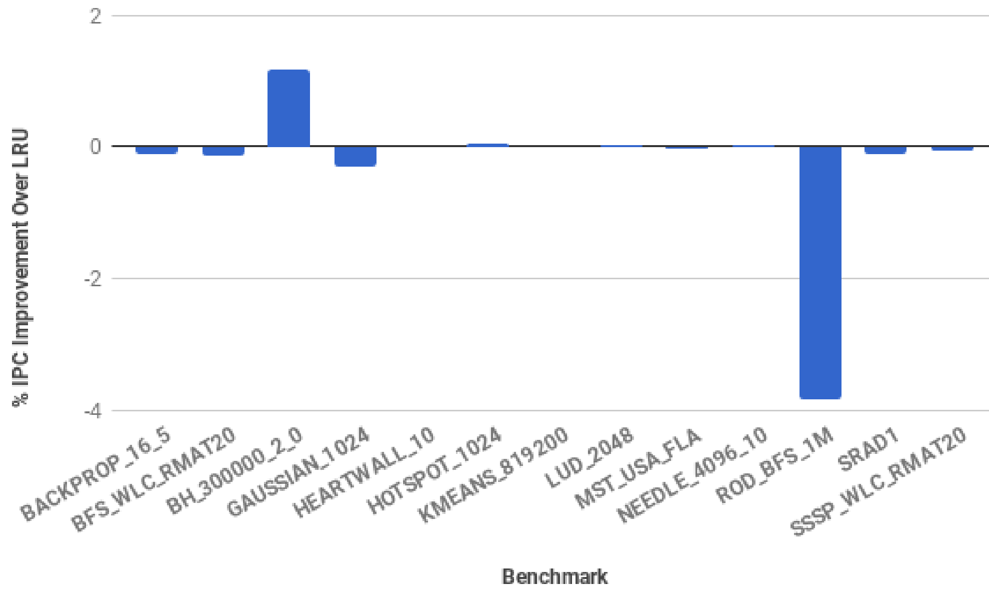

Figure 5.4: IPC improvement over LRU when cleaning the least recently used line across all sets.

Our idea of precleaning certainly needs more evaluation and exploration of prediction mechanisms beyond what is provided in this report. We believe with more time and better domain knowledge we could make real performance improvements. Though it remains uncertain if these performance improvements would be significant enough to warrant the cost of an entirely new prediction unit in the L2 cache. This prediction unit could be costly as it needs to evaluate precleaning candidates, clean lines, and predict bandwidth usage on a regular basis.

We believe that, right now, precleaning isn't a viable approach for

improving memory system performance on GPUs. While we can currently achieve small improvements in performance, the problem space is ill defined and the predictors will be hard to tune. If GPUs become more starved for bandwidth in the future we believe this would give us the headroom required to make precleaning worth the effort.

Chapter 6

Future Work and Remarks

In this section we touch on future work and next steps for both Hawkeye and precleaning. We also talk about lessons learned from the research and how the author can improve their research in the future. Furthermore, we give suggestions for anyone hoping to do similar research on GPUs.

6.1 Future Work

While the results from the previous section show promise for both Hawkeye and precleaning applied to GPUs, there's still considerable work before either of these ideas can be fully evaluated. Hawkeye appears to give decent performance improvements, but it needs more tuning of the predictors and features. Precleaning also shows promising headroom, but we would like to explore better predictors and more fitting metrics for bandwidth usage and data divergence.

Hawkeye gives good results, but the performance impact of the additional meta-data store still needs to be fully evaluated, and we need to explore potentially better OPTgen training features. Furthermore, it is unclear whether Hawkeye will perform well if applied to both the L1 and L2 caches simultaneously,

and we would also like to see if we could apply warp sampling techniques as seen in APCM. Given the results in the previous section, we are unsure whether or not the performance gains are worth the cost of implementation when APCM seems to give better results. Finally, we would like to evaluate a larger set of benchmarks. Jeff Diamond's thesis uses the Parboil benchmark suite in addition to Rodinia, noting that benchmarks like pns show large usage of the cache [10]. We believe Diamond's evaluation of cache intensity and reuse across benchmarks can help us select a more representative selection of benchmarks in the future.

Precleaning, as mentioned before, shows promise, but it requires a significant amount of additional work to fully evaluate. Currently one of the limiting factors to additional research is a lack of well defined metrics that can easily measure performance improvements. IPC improvements and memory bandwidth usage are both fairly coarse grained metrics. We would like to find a metric, for instance, that gives a clear indicator of memory stalls and data divergence in each benchmark. Being able to measure the level of data divergence would allow us to see headroom and improvements for a precleaner, rather than bandwidth which doesn't give us an idea of our headroom.

6.2 Remarks and Lessons Learned

We believe that the results of this research will be useful, but we are slightly disappointed by the amount of progress made and how we left certain avenues of research unexplored. One of the biggest hurdles to overcome

was understanding both the landscape of GPU programming and the tooling provided for GPU simulation.

Our suggestions to others hoping to pick up from this research or study other GPU architecture topics: work with a known environment, use a powerful, highly parallel machine, and explore alternate tooling. GPGPUsim is quite old at the time of writing, and setting up the simulator reliably will only get harder as time goes on. Finding the appropriate versions of gcc, g++, and CUDA can be difficult on modern operating systems (Ubuntu 16.04 LTS). For this reason, if one decides to use GPGPUsim, we suggest working with a known environment that has been successfully used for GPU research in the past. We also suggest using a highly parallel machine. GPGPUsim will only use a single core per simulation, but each simulation can take upwards of 24 hours with no option for skipping warmup periods. One can save a lot of time by running all relevant benchmarks in parallel over the course of a day.

Finally, as mentioned before, GPGPUsim hasn't received substantial updates in almost 4 years. GPGPUsim was vital for getting the results given in this paper, but in the future it will only get further from state of the art GPU architecture. Thus, we suggest exploring simulators that are more regularly updated and maintained.

Chapter 7

Conclusion

In this research report we evaluated the cache replacement policy Hawkeye on GPUs and provided coarse headroom for the idea of precleaning. We believe both of these ideas can provide benefits to GPU performance, but current implementations are fairly situational and require the code being run to be sensitive to caching performance and bandwidth usage. We conclude that Hawkeye will provide the most benefit when applied at the L1 cache with just PC as the feature for OPTgen. However, from our performance impact experiments, we believe there's more room for improvement especially with the L2 cache replacement policy. Furthermore, our idea of precleaning requires more sophisticated predictors and metrics before it can be fully evaluated, and we are unsure if this effort is worth the small increases in performance indicated by our headroom study.

Appendices

Appendix A

Hawkeye Experimental Results

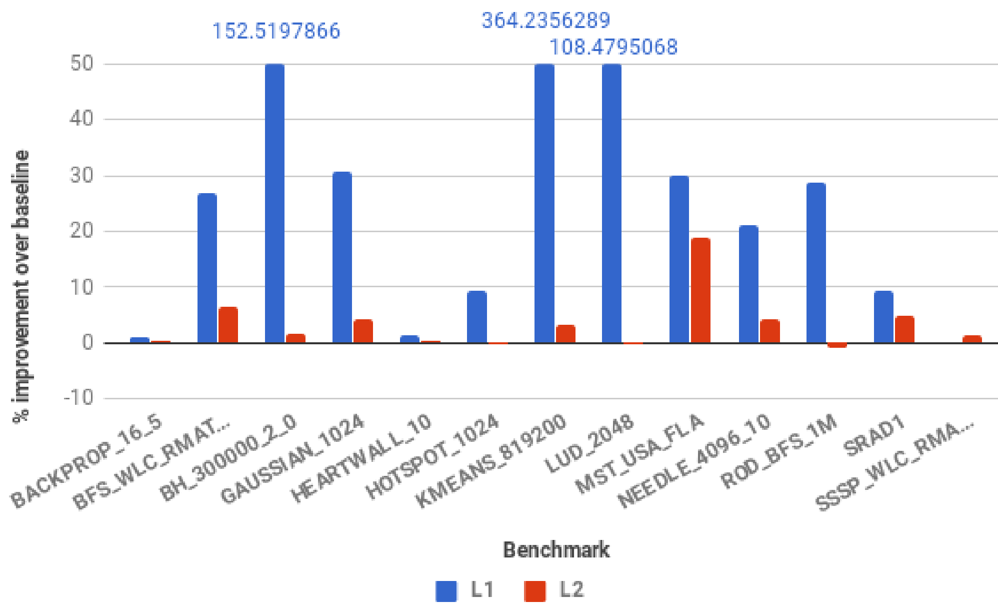

Figure A.1: Cache sensitivity for both L1 and L2 caches. These numbers are percentage improvements over in IPC over baseline when quadrupling the size of each cache. Replacement for all cases is done with LRU.

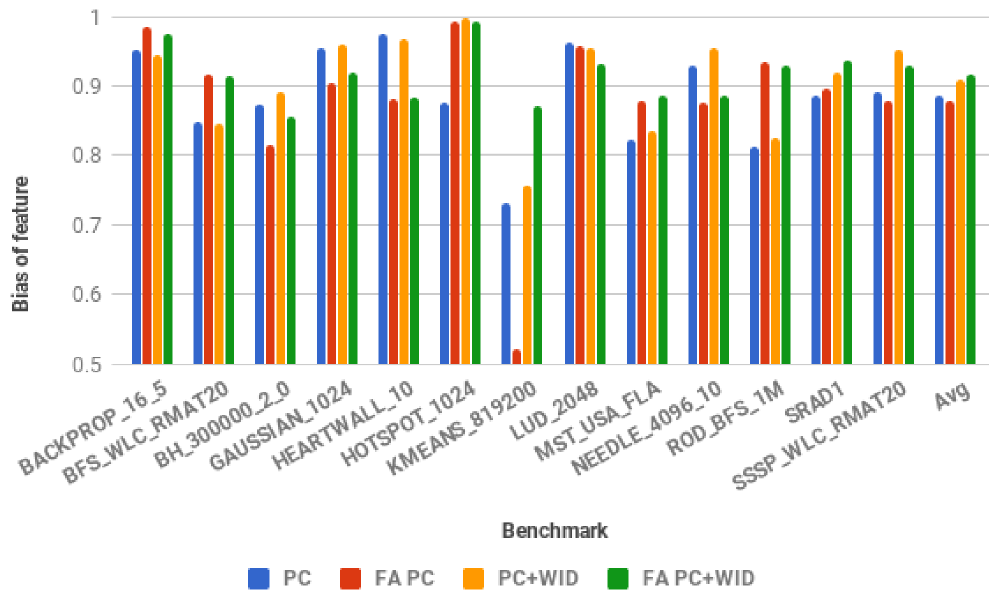

Figure A.2: Average per-feature prediction bias for L1 caches. The bias denotes the percentage of truth values that agree with our predicted value of cache friendly or unfriendly. Here 50% would be equivalent to a random prediction. The final column denotes our average per-feature bias across all benchmarks.

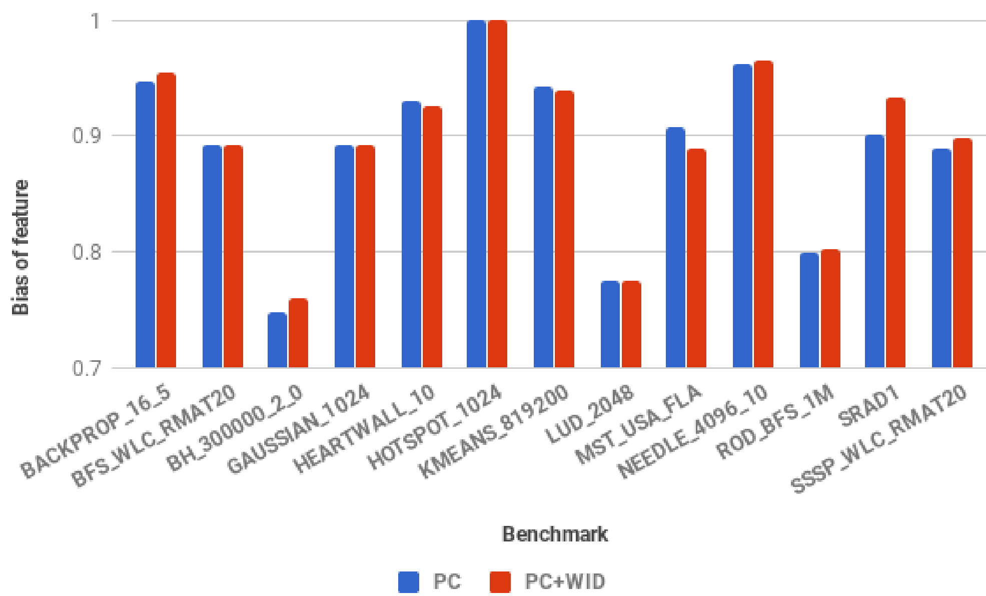

Figure A.3: Average per-feature prediction bias for L2 caches.

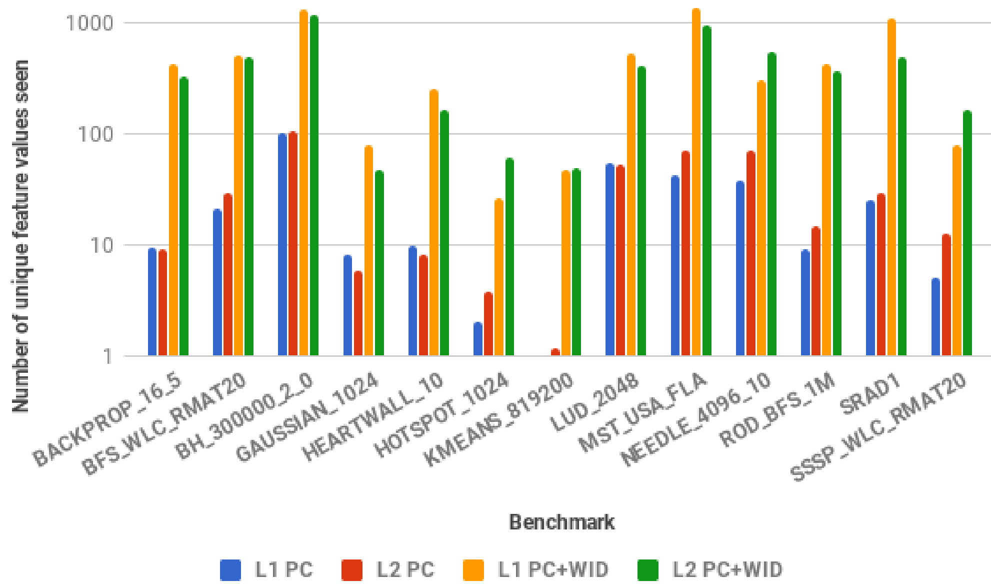

Figure A.4: Average unique values per feature at L1 and L2 caches.

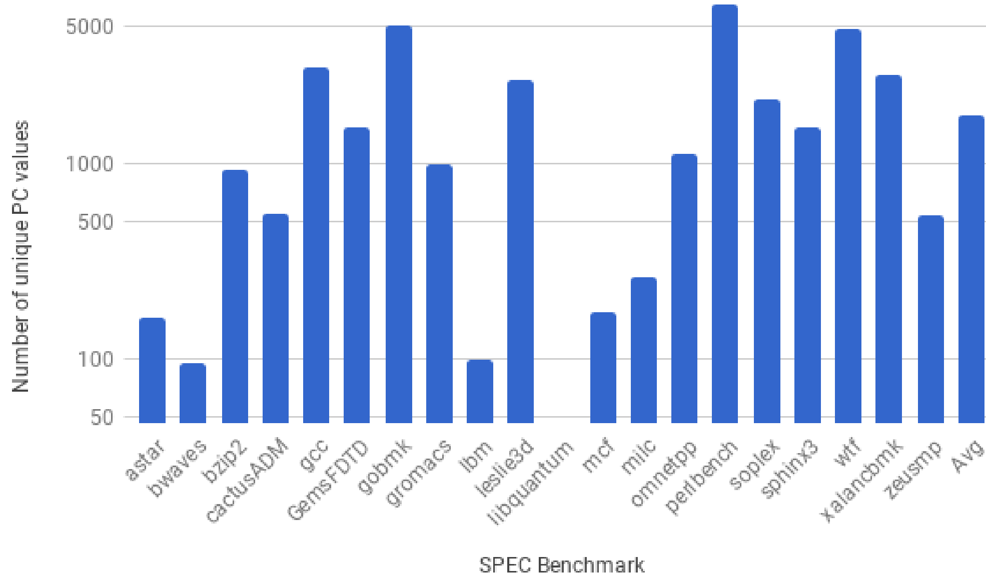

Figure A.5: Number of unique PC values seen on CPU SPEC Benchmarks. Notice that these numbers are roughly on the same order of magnitude as the number of PC+WID hashed values we see on GPU.

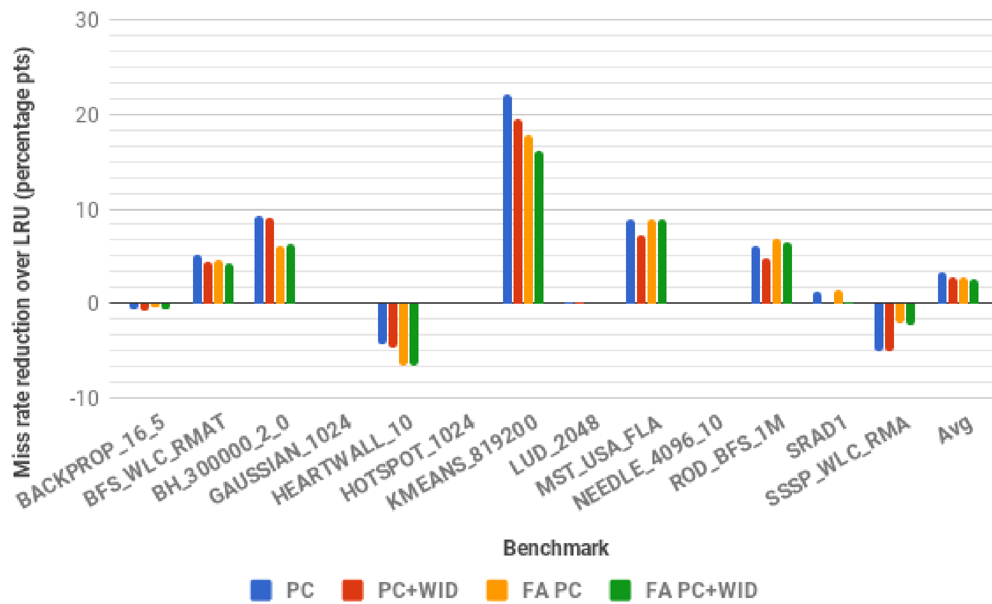

Figure A.6: Average miss rate improvement over LRU for L1 caches. This improvement is measured in percentage points over the miss rate observed with LRU.

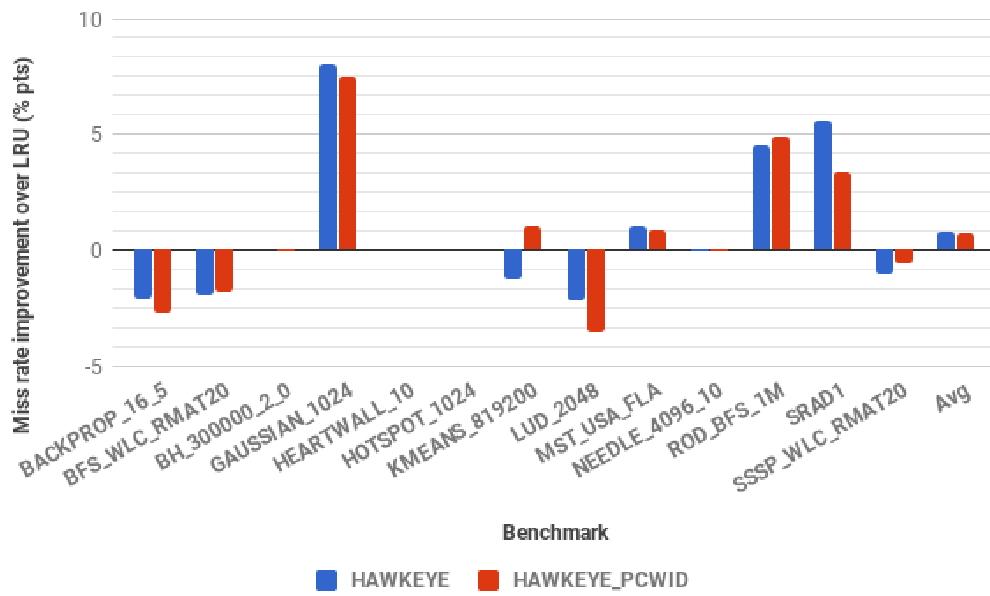

Figure A.7: Average miss rate improvement over LRU for L2 caches. This improvement is measured in percentage points over the miss rate observed with LRU.

Bibliography

[1] Alaa R Alameldeen, Aamer Jaleel, Moinuddin Qureshi, and Joel Emer. 1st jilp workshop on computer architecture competitions (jwac-1) cache replacement championship, 2010.

[2] Aaron Ariel, Wilson WL Fung, Andrew E Turner, and Tor M Aamodt. Visualizing complex dynamics in many-core accelerator architectures. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 164–174. IEEE, 2010.

[3] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[4] Nathan Beckmann and Daniel Sanchez. Maximizing cache performance under uncertainty. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 109–120. IEEE, 2017.

[5] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[6] Laszlo A Belady, Robert A Nelson, and Gerald S Shedler. An anomaly in space-time characteristics of certain programs running in a paging

machine. *Communications of the ACM*, 12(6):349–353, 1969.

[7] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[9] Asit Dan and Don Towsley. *An approximate analysis of the LRU and FIFO buffer replacement schemes*, volume 18. ACM, 1990.

[10] Jeffrey Robert Diamond. *Designing on-chip memory systems for throughput architectures*. PhD thesis, 2015.

[11] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420. IEEE Computer Society, 2007.

[12] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation: Efficient mimd control flow on simd graphics hardware. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):7, 2009.

[13] David B Glasco, Peter B Holmqvist, George R Lynch, Patrick R Marchand, James Roberts, and John Edmondson. System and method for cleaning dirty data in an intermediate cache using a data class dependent eviction policy, August 14 2012. US Patent 8,244,984.

[14] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[15] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 78–89. IEEE, 2016.

[16] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[17] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. Access pattern-aware cache management for improving data utilization in gpu. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 307–319. ACM, 2017.

[18] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ACM SIGARCH Computer Architecture News*, volume 29, pages 144–154. ACM, 2001.

[19] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 222–233. IEEE, 2008.

[20] Steven Pigeon. Pairing function. `http://mathworld.wolfram.com/PairingFunction.html`.

[21] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007.

[22] Kim Schuttenberg and R Frank O'Bleness. Systems and methods for writing data from a caching agent to main memory according to a pre-clean criterion, September 22 2015. US Patent 9,141,543.