Ctrl-C: Instruction-Aware Control Loop Based Adaptive Cache Bypassing for GPUs

Shin-Ying Lee and Carole-Jean Wu
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
{lee.shin-ying,carole-jean.wu}@asu.edu

Abstract—The performance of general-purpose graphics processing units (GPGPUs) is often limited by the efficiency of the memory subsystems, particularly the L1 data caches. Because of the massive multithreading computation paradigm, significant memory resource contention and cache thrashing are often observed in GPGPU workloads. This leads to high cache miss rates and substantial pipeline stall time. In order to improve the efficiency of GPGPU caches, we propose an instruction-aware control loop based adaptive cache bypassing design (Ctrl-C). Ctrl-C applies an instruction-aware algorithm to dynamically identify per-memory instruction cache reuse behavior. Ctrl-C then adopts feedback control loops to bypass memory requests probabilistically in order to protect cache lines with short reuse distances from early eviction. GPGPU-sim simulation based evaluation shows that Ctrl-C improves the performance of cache sensitive GPGPU workloads by 41.5%, leading to higher cache and interconnect bandwidth utilization with only an insignificant 3.5% area overhead.

I. INTRODUCTION

Graphics processing units (GPUs) are widely used in high performance computing systems to accelerate general-purpose parallel computation. General-purpose GPUs (GPGPUs) achieve a significant throughput by processing massive number of concurrent threads in the single instruction multiple thread (SIMT) manner. Similar to traditional chip multiprocessors (CMPs), modern GPUs are also equipped with caches to reduce the average memory access latency and the interconnect bandwidth requirement. However, because caches are often shared across all concurrent threads, the per-thread cache capacity is often too small for GPUs. Threads contend for the cache storage with each other, resulting in cache thrashing, i.e., cache lines are frequently swapped in/out without receiving any reuse. Consequently, GPUs do not utilize caches efficiently, L1 data caches, in particular.

The cache inefficiency in GPUs raises two critical problems which often limit the performance of GPUs. First, due to the cache thrashing problem, many GPGPU applications have high data cache miss rates. GPU caches are not able to effectively reduce the average memory access latency, leading to extra pipeline stalls. Second, a large amount of adjacent data elements brought into caches with the demanded data is never referenced before being evicted. This injects additional interconnect traffic that can increase the queuing latencies in the interconnect. Because of the unnecessary data traffic, using caches to preserve spatial localities may significantly degrade the performance in some GPGPU workloads [1].

To alleviate the degree of cache thrashing in GPUs, many prior works proposed to apply cache bypassing techniques. A widely used approach is to employ compilers to perform offline analysis and identify data that are unlikely to receive any reuse in the near future [1], [2], [3]. However, the compiler-based algorithms are not flexible for input dependent applications. Aside from the compiler-based schemes, a number of prior works proposed to use additional hardware components to count and predict the reuse distances of cache lines at runtime [4], [5]. The reuse distances of GPGPU cache lines can be extremely long and exhibit a disperse distribution. It is challenging to accurately predict reuse characteristics of GPGPU cache lines with limited storage requirement. These dynamic prediction algorithms, thus, require a large number of counters and incur significant implementation overhead.

To tackle the cache inefficiency problem in GPUs, we propose a low implementation overhead design: Instruction-Aware Control Loop Based Adaptive Cache Bypassing (Ctrl-C) to accurately predict the cache reuse behavior in the granularity of an instruction and statistically bypass memory requests to prevent cache lines from early eviction. Ctrl-C first applies an instruction program counter (PC) based algorithm to identify the cache reuse behavior. Ctrl-C then employs feedback control loops to learn the per-instruction reuse patterns. By doing so, a part of an instruction’s active working set is retained in the cache, receiving reuses. Furthermore, since data inserted in the cache starts getting reuses, the reuse predictor learns this updated re-reference information in the control loop at runtime and bypasses requests accordingly. Eventually, it reaches a stable state by settling at the optimal bypassing aggressiveness. Our evaluation results show that Ctrl-C can improve the performance of GPGPU workloads by an average of 41.5% over the baseline configuration while the state-of-the-art PC-based adaptive bypassing scheme improves the performance by 22.6% [4], [6]. Overall, Ctrl-C’s performance speedup is close to that of doubling the L1 data cache capacity with only 3.5% hardware overhead.

II. BACKGROUND AND MOTIVATION

A. GPU Data Cache Inefficiency

GPUs execute programs in a massive multithreading manner. Thousands of threads run simultaneously and compete for hardware resources, such as the L1 data caches, L2 caches, and the interconnect bandwidth. This makes cache capacity and interconnect bandwidth critical resources for GPUs.
Access over the Baseline

**Algorithm 1** An example of GPGPU kernel source code from KMN.

```plaintext
1: function KERNEL(out[], in[], int m)
2:   for i in [0 : m] do
3:     tid = the unique thread id from 0 to (n - 1)
4:     v ← in[tid * m + i]
5:     performing computations on v
6:     out[tid * m + i] ← v
7:   end for
8: end function
```

Figure 1 shows the performance sensitivity of GPGPU applications to the L1 data cache capacity. The x-axis represents the wide range of GPGPU applications studied in this paper (more methodology detail is given in Section IV-B) whereas the y-axis represents the speedup normalized to the baseline 16kB L1 data cache configuration. We observe that a large number of GPGPU applications—the cache sensitive (CS) workloads—gain a significant speedup with the increase in the L1 data cache capacity. When the L1 data cache size is quadrupled from the baseline 16kB configuration to 64kB, an average of 2.29x performance speedup is gained for the CS GPGPU applications. This indicates that a significant performance improvement can be gained if the L1 data cache capacity is increased or is managed more efficiently, such that the large working sets common in GPGPU workloads can be accommodated more effectively.

Moreover, from Figure 1, we also notice that there are a number of GPGPU applications benefiting from turning off the L1 data caches completely, e.g., MM, PRK and KMN. This is because in these GPGPU applications, a large amount of data elements adjacent to the demanded data in a cache line are brought into the cache but are not reused during its cache lifetime before being evicted from the cache (early eviction). Thus, spatial locality is not exploited efficiently, resulting in poor cache line utilization that wastes interconnect bandwidth and introduces additional queuing latency [1], [7]. Since simply increasing cache capacities and interconnect bandwidth to speed up the execution of GPGPU applications costs tremendous storage overhead, it is an impractical solution. A more sophisticated cache management approach is needed to improve the resource utilization of the memory subsystem in GPUs.

**B. GPU Data Cache Reuse Behavior**

In order to understand the root-cause of inefficiency in GPU caches, we delve deeper to investigate the cache access behavior of GPGPU applications. We use the cache access pattern directly from a cache sensitive application from the Rodinia benchmark suite [8]—k-means clustering (KMN)—to illustrate the cache thrashing behavior generated as a result of the SIMT execution paradigm.

Algorithm 1 shows the kernel source code from KMN. This kernel iteratively reads data from the `in` array and performs computations on array elements with `n` concurrent threads, where each thread works on `m` array elements. At each iteration, array elements (`in[i]`, `in[m+i]`, `in[2*m+i]`, ..., `in[n*m+i]`) are accessed sequentially by different threads and these array elements are mapped to multiple cache lines as shown in Figure 2. Since `n` is usually very large for GPGPU workloads, we would expect to see an access pattern like \((a_0, a_1, ..., a_k)^N\) in a single cache set, where \(a_i\) represents a unique access to the cache set and \(N\) represents the number of repeats. When \(k\) (or equivalently the reuse distance) is greater than the cache set associativity \((S)\), cache thrashing occurs. Cache lines are evicted before they are re-referenced and all memory accesses result in cache misses. Because GPUs process thousands of threads in parallel with large working sets, a similar cache access pattern with a large \(k\) is observed in GPGPU workloads. Cache thrashing is a primary bottleneck limiting the performance of GPUs.
For such thrashing access behavior, it has been proved that inserting exactly $S$ cache lines to a particular cache set and bypassing all other memory requests can achieve an optimal cache hit rate [9], [10]. However, it is virtually impossible to identify $k$ in advance because of the diverse behavior of GPGPU workloads. Figure 3 shows the distribution of L1 data cache reuse distances ($k$) for GPGPU applications. The stacked bars show the distribution of reuse distances whereas and the black curve indicates the median reuse distance.

Figure 3 provides three important insights for cache access behavior of GPGPU workloads. First, since a common configuration for the GPU L1 data cache is 4 or 8-way set associative, a huge portion of cache lines will never be reused during its lifetime. Second, GPGPU applications have a diverse reuse behavior. For example, PVR has a long median reuse pattern whereas ELL has a short median reuse pattern. Finally, the reuse distance can be extremely long and the distribution is dispersed within an application. For instance, in PVR, 29% of cache lines have reuse distances longer than 128 while 17% of cache lines have reuse distances less than 4 and are expected to receive cache hits. It is difficult to design a static cache management policy that can achieve the optimal hit rate for all GPGPU applications.

Figure 4 shows an example of the distribution of L1 data cache reuse distances with different memory instructions from BFS. Figure 4 reveals that cache lines inserted by the same load instructions often have a similar reuse pattern. For example, most cache lines inserted by PC_0, PC_1, PC_6, and PC_8 in BFS have short reuse distances. On the other hand, the cache lines from the other memory instructions have long reuse distances that are likely to incur cache misses. This indicates that the unique program counter (PC) values of memory instructions can be a signature to predict and identify the behavior of cache lines.

C. Cache Bypassing for GPUs

Cache bypassing is a common technique to mitigate cache thrashing. Nevertheless, rather than bypassing all requests, bypassing only a selective portion of memory requests can achieve a better cache hit rate and execution time speedup [9], [10]. While we have identified that different memory instructions have different cache reuse patterns, we find that, instead of bypassing all memory references from an instruction with long reuse behavior, bypassing only a selective portion of the memory references offers additional performance gain.

Figure 5 shows the application execution time speedup with different per-instruction bypassing probabilities. The $x$-axis represents memory instructions from different GPGPU applications and the $y$-axis represents the speedup normalized to no cache bypassing. Each bar in Figure 5 represents a bypassing probability—$(1 - \frac{1}{N})$—imposed on the important memory instructions in three applications, BFS, ELL, and KMN. With a greater $N$, a larger portion of memory requests are bypassed stochastically whereas, with a smaller $N$, a larger portion of memory requests are inserted into the cache.

Not all memory instructions benefit from the same degree of cache bypassing. The optimal bypassing probability varies from instruction to instruction, and from application to application. The optimal $N$ for ELL’s PC_3 is 3 and for BFS’s PC_4 is 7 (or bypassing all) whereas the optimal $N$ for KMN’s PC_0 is 5. Moreover, not all instructions benefit from bypassing, as illustrated by BFS’s PC_6. To capture the significant performance improvement potential with bypassing, a design must be able to predict the reuse patterns dynamically and adopt a variable bypassing probability on the basis of instructions.

III. CTRL-C: INSTRUCTION-AWARE CONTROL LOOP

A. Overview of the Ctrl-C Design

In order to improve the efficiency of GPU caches, an Instruction-Aware Control loop based adaptive Cache bypassing design, called Ctrl-C, is devised. By taking the insights from Section II, Ctrl-C (1) dynamically learns and predicts the reuse patterns of instructions in GPGPU programs, and (2) statistically bypasses memory requests from the L1 data caches for instructions that generate requests with a low likelihood of reuse. Ctrl-C applies feedback control loops to train the entries of an instruction reuse prediction table (Section III-B) with the reuse history of evicted cache lines. When the fraction of zero-reuse cache lines inserted by a particular instruction is high, Ctrl-C starts bypassing memory requests from this instruction and increases its bypassing aggressiveness until a stable state.
is reached—cache lines inserted with the instruction start receiving hits (Section III-C). By doing so, a portion of the instruction’s working set is retained in the cache, leading to increased cache utilization.

B. The Instruction Reuse Prediction Table (iReuse Table)

The goal of the instruction reuse prediction table, or iReuse Table, is to predict whether a cache line inserted by an instruction will receive future reuses and to determine if a memory request should be bypassed from the cache. The intuition is that if a cache line inserted by a particular instruction receives hits, future cache insertions by the same instruction will similarly receive hits. To explicitly correlate the reuse patterns to the insertion instructions, we implement iReuse Table as a hash table indexed by the lower bits of an instruction PC. iReuse Table is a simple hash table, where each table entry is connected to a feedback control loop III-C.

iReuse Table is trained with the reuse histories of cache lines in the L1 data caches. To track the reuse histories, Ctrl-C augments the meta data field of a cache line by two additional parameters—a 1-bit reuse parameter and a 7-bit insertion instruction parameter. The reuse bit is set to 0 when a cache line is first inserted into the cache and is set to 1 when it receives a cache hit. The insertion instruction is set to the lower 7-bit of the instruction PC that causes the cache insertion. When a cache miss occurs and a new cache line is inserted into the cache, an eviction candidate is selected based on the underlying cache replacement policy. The reuse history of the evicted cache line—(insertion instruction, reuse)—is used to train the corresponding iReuse Table entry.

C. Feedback Control Loop for Modulating Per-Instruction Bypass Aggressiveness

The goal of the per-instruction feedback control loops in iReuse Table is to learn the reuse distances and to modulate the aggressiveness of per-instruction bypassing until a portion of the instruction’s inserted data is retained in the cache. While iReuse Table separates instructions that generate memory requests with high and low likelihood of reuses, for this instruction-based reuse learning and prediction mechanism to perform well, a design must handle bypassing appropriately.

The feedback loop controller applies four counters to keep track of the cache utilization behavior: a 3-bit AGG saturating counter, a 7-bit BYP counter, a 10-bit ZERO counter, and a 10-bit INSERT counter. AGG represents the aggressiveness of bypassing and controls the probability of cache line insertion and BYP counts the total bypassing requests. AGG and BYP are used to regulate the aggressiveness of bypassing whereas ZERO and INSERT are used to keep track of the cache utilization. Specifically, cache lines of an instruction have a probability of $1/\text{AGG}$ to be inserted into the cache.

D. Ctrl-C Algorithm

Figure 6 illustrates the design structure of Ctrl-C. Upon a cache miss, the corresponding iReuse Table entry is accessed for a decision on either inserting or bypassing the cache line. If a cache line is to be bypassed, the instruction’s corresponding controller increments BYP and bypasses the cache line. If a cache line is to be inserted into the cache, an eviction candidate is selected and its reuse history (insertion instruction, reuse) is used to train the iReuse Table. The controller increments INSERT to keep track of the number of insertions that have occurred thus far, and if the evicted cache line’s reuse bit is 0, indicating that it has not received a reuse during its cache lifetime, the controller also increments ZERO to keep track of the number of zero-reuses. Then, the per-instruction controller for the corresponding instruction increments BYP.
Algorithm 2 The operations of Ctrl-C at a cache miss – determining bypassing or insertion.

1: function \texttt{ATMISS}(memRequest)
2: \> \(\triangledown\) determine whether bypassing or not
3: \> ctrl \(\leftarrow\) 
4: \> \(\triangledown\) bypassing the request
5: \> \texttt{bypass}(memRequest)
6: \> ctrl.BY \(\leftarrow\) ctrl.BY \(+\) 1
7: \> \(\triangledown\) inserting a new cache line
8: \> evictedLine \(\leftarrow\) insert(memRequest)
9: \> \texttt{atEviction}(evictedLine)
10: \> ctrl.BY \(\leftarrow\) 0
11: \> \texttt{end if}
12: \> \texttt{end function}

Algorithm 3 The operations of Ctrl-C at a cache line eviction – updating the bypassing aggressiveness.

1: function \texttt{ATEVICT}(evictedLine)
2: \> \(\triangledown\) update iReuse table
3: \> ctrl \(\leftarrow\) iReuse[evictedLine.insertPC].ctrl
4: \> ctrl.INSERT \(\leftarrow\) ctrl.INSERT \(+\) 1
5: \> \(\triangledown\) based on whether
6: \> \texttt{end if}
7: \> \texttt{end function}

with a bypassing decision. If a cache line gets evicted, iReuse also increments \texttt{INSERT} and updates \texttt{ZERO} based on whether the evicted cache line has received a reuse or not during the line’s lifetime in the cache.

The feedback controller learns the optimal bypassing aggressiveness by periodically examining the current cache utilization, namely, the number of zero-reuse cache lines out of the total inserted cache lines. Specifically, the controller aims to keep the zero-reuse portion within a target range \((\text{THRESHOLD}_H \text{ and } \text{THRESHOLD}_L)\) for cache performance improvement. If the zero-reuse fraction is greater than a certain target threshold \((\text{THRESHOLD}_H)\), the controller switches to bypassing aggressively by modulating \texttt{AGG}. Specifically, during a time period, if \texttt{INSERT} is lower than the threshold \((\text{THRESHOLD}_L)\), the controller decrements \texttt{AGG} by 1 to bypass memory requests less aggressively. When \texttt{ZERO} settles between \texttt{THRESHOLD}_L and \texttt{THRESHOLD}_H, the controller is in a stable state with the optimal degree of bypassing. Consequently, the cache retains a portion of the instruction’s inserting data probabilistically and its utilization is more efficient. Algorithm 2 and 3 describe the pseudo-code of Ctrl-C.

E. Ctrl-C Design Parameters

The target thresholds should provide the following features. First, the number of zero-reuse cache lines or \texttt{ZERO} should be as low as possible in order to eliminate the cache thrashing problem and guarantee no early eviction. Second, the controller should be able to detect and recover from an incorrect prediction. When \texttt{ZERO} is low, it is likely that the controller bypasses memory requests too aggressively and loses the opportunity to recover cache lines that are mistakenly bypassed. Therefore, \texttt{THRESHOLD}_L should be slightly higher than zero to leave room for incorrect prediction detection and recovery. Additionally, \texttt{THRESHOLD}_H represents the number of zero-reuse lines allowed, it should be small to keep \texttt{INSERT} low. Finally, the bypassing aggressiveness should reach a stable state quickly to attain as much performance gain as possible. For our study, the target threshold range, \([\text{THRESHOLD}_L,\text{THRESHOLD}_H]\), is set as \([0.1, 0.4]\).²

Another critical parameter in the feedback controller design is the sampling period. The feedback controller uses the number of evictions as the unit of the sampling period to evaluate the status of cache utilization and to update bypassing aggressiveness accordingly. However, when bypassing is frequent with a high \texttt{AGG} value, the frequency of cache evictions decreases and causes the controller a longer learning time. Thus, the bypassing aggressiveness may not get updated quickly enough. In order to adjust the learning rate according to the bypassing/insertion rate, we adjust the sampling period based on the bypassing aggressiveness as follows (Equation 1).

\[ \text{SAMPLE}_\text{PERIOD} = \text{BASE}_\text{PERIOD} >> \text{AGG} \]

F. Hardware Implementation Overhead

Ctrl-C has a low implementation overhead. Its implementation overhead includes the feedback controllers, and the two additional metadata fields per cache line. Overall, with the baseline 16kB data cache (32-set, 4-way set associative), Ctrl-C needs only 608 bytes of additional storage. Compared to the baseline 16kB cache, this corresponds to 3.5% storage overhead. As this paper later shows, this extra storage brings a significant 41.5% performance improvement for cache sensitive GPU workloads (Section V-A).

IV. METHODOLOGY

A. Simulation Infrastructure

To evaluate the performance improvement of the Ctrl-C, we use GPGPU-sim (version 3.2.2) [15]. GPGPU-sim is a cycle-level simulator that simulates a GPGPU and supports NVIDIA CUDA [16] as well as the PTX ISA [17]. We implement the proposed Ctrl-C design on top of GPGPU-sim and run with the default configuration to simulate NVIDIA

²The parameters are chosen empirically through sensitivity tests.
Fermi GTX480 [18]. As a comparison, we also implement the PC-based Adaptive Bypassing, which uses per-instruction confidence counters to predict the reuse patterns [6], [4]. When detecting cache lines inserted by an instruction do not have any reuse, it bypasses all memory requests generated by this particular instruction. Table I and II show the configurations of the simulation setup and the control loop parameters for Ctrl-C in detail.

B. GPGPU Benchmarks

We select a wide range of GPGPU applications from the Mars [13], NVIDIA SDK [11], Pannatia [14], and Rodinia [8], [12] benchmark suites to represent the diverse behavior of GPGPU workloads. Based on the performance sensitivity to the cache size increase from the baseline 16kB to 64kB, we classify these applications into two categories: (1) Cache-Sensitive (CS) applications which achieve a speedup greater than 1.2x with the 64kB L1 data cache and (2) Non-Cache-Sensitive (NS) applications which has less than 1.2x speedup with the cache size increase from the baseline.

V. Experimental Result and Analysis

A. Overall Performance Improvement

Our simulation results show that compared to the baseline 16kB L1 data cache configuration, the proposed Ctrl-C improves GPGPU performance by an average of 41.5% for CS applications and 20.7% across all applications as shown in Figure 7. With Ctrl-C, all CS applications, except FLD, can obtain more than 1.1x speedup and the performance gain can be as high as 2.39x (KMN). We notice that FLD does not achieve a good speedup. This is because FLD does not have a high fraction of zero-reuse lines (Section V-C). Ctrl-C does not bypass any memory request and hence the performance is the same as the baseline.

The PC-based Adaptive Bypassing scheme improves the performance of the cache sensitive workloads by an average of 22.6%, that is 19% lower than that of Ctrl-C. This is because, instead of bypassing cache lines of an instruction probabilistically, when the PC-based Adaptive Bypassing scheme detects a portion of cache lines of an instruction do not receive reuse hits, it starts bypassing all memory references from this instruction. As a result, this design loses the opportunity to
learn and capture the cache lines that can receive potential reuses once it has learned the per-instruction reuse history. Moreover, it is difficult to detect an incorrect prediction in such a design if it bypasses all requests and therefore, with the PC-based Adaptive Bypassing, an application may instead experience performance degradation, e.g., FLD. Overall, Ctrl-C achieves a significant speedup that is close to using a double sized (32kB) L1 data cache, and outperforms the PC-based Adaptive Bypassing.

B. MPKI and Interconnect Traffic Reduction

To investigate where the significant performance improvement comes from, we next investigate the ability of Ctrl-C to reduce the number of misses per kilo instructions (MPKI) and interconnect traffic.

Figure 8 shows the normalized MPKI with Ctrl-C for all CS applications. For CS applications, the working set is typically much larger than the data cache capacity. A large amount of data is evicted from the data caches before receiving any reuse. Therefore, the performance of CS applications is mainly restricted by the cache thrashing problem. However, Ctrl-C adaptively protects cache lines from early eviction by bypassing a part of the memory requests and thereby some cache misses turn into hits. Overall, with Ctrl-C, the MPKI of CS applications is reduced 9.9% that effectively translates into 41.5% performance speedup.

The interconnect bandwidth is another critical resource limiting GPU performance. Figure 9 shows the interconnect traffic reduction with Ctrl-C. Ctrl-C is able to achieve 43.7% traffic reduction for CS applications compared to the baseline. It filters out data traffic by reducing the MPKI and the zero-reuse data elements.

C. Fraction of Zero-reuse Lines

While CS applications have a significant portion of zero-reuse lines due to the severe cache thrashing problem in the baseline 16kB L1 data caches, Ctrl-C can effectively reduce the number of zero-reuse lines. Figure 10 shows the percentage of zero-reuse lines with Ctrl-C. For CS applications, the percentage of zero-reuse lines reduces significantly by an average of 48.7%. We notice that PF still has a high fraction of zero-reuse lines with Ctrl-C. This is because the kernel execution time of PF is too short. Ctrl-C does not have sufficient time to learn the optimal bypassing aggressiveness for this particular application. Overall, with Ctrl-C, the average fraction of zero-reuse lines for all applications reduces to 44.1%.

VI. RELATED WORK

Many cache management policies have been proposed to mitigate cache thrashing in CPUs, such as [19], [20], [21]. However, due to the small per-thread cache capacity, employing these algorithms is not able to accurately predict the data reuse patterns in GPUs.

To improve the cache hit rate with a thrashing access pattern, Qureshi et al. [10] proposed the BIP policy to bypass memory requests probabilistically. However, the bypassing probability of BIP is fixed across all memory instructions and is trained off-line. BIP is not flexible for different applications. Wu et al. [6] proposed a PC-based reuse prediction algorithm to predict the cache reuse behavior and use the prediction to guide cache insertion positions. While SHiP can be extended for cache bypassing, it does not use different aggressive levels for bypassing cache lines. In contrast, the proposed Ctrl-C scheme trains the per-instruction bypassing aggressiveness adaptively and achieve a better performance improvement.

In order to alleviate the resource contention in GPUs, many cache bypassing algorithms have been proposed. Jia et al. [22] designed a FIFO queue (MRPB) to reorder requests targeted at reducing inter-warp contention. Additionally, MRPB bypasses requests if intra-warp contention is detected. Chen et al. [23] designed an adaptive resource management scheme that monitors cache contention and interconnect congestion. If the degree of cache contention or bandwidth demand is too high, memory requests will be bypassed. However, MRPB and the adaptive resource management designs do not distinguish reuse patterns among memory requests. Cache lines with near reuse distances may be bypassed, losing an opportunity to
improve the cache hit rate. In contrast, Ctrl-C only bypasses requests having long reuse patterns.

Xie et al. [2] and Liang et al. [3] modified compilers to analyze GPGPU applications and guide GPUs to bypass data which are unlikely to receive reuse. However, the compiler based schemes are not able to predict the reuse behavior of input dependent applications. Tian et al. [4] proposed the PC-based Adaptive Bypassing that uses confidence counters to predict zero-reuse lines and bypasses all requests if detecting cache lines will not receive any reuse. Instead of bypassing all requests, Ctrl-C bypasses more intelligently by inserting only a selective portion of requests which mitigate the degree of cache thrashing. Furthermore, Ctrl-C enables per-instruction bypassing probabilities. By doing so, Ctrl-C can more effectively learn the reuse patterns of instructions in GPUs. Lee et al. [24] designed a technique called CCBP to predict cache reuse behavior and prioritize cache resources for cache lines required by critical warps. While CCBP targets to accelerate the critical warp execution, the proposed Ctrl-C can further improve the performance of GPUs by reducing the overall MPKI and execution time of GPUs for all warps. Li et al. [5] suggested adding additional tag array entries to track the data reuse patterns. Nevertheless, the reuse distance can be extremely long for GPUs. It is challenging to accurately predict the data reuse patterns with a limited number of tag array entries. In contrast, Ctrl-C uses feedback controllers to learn the reuse pattern with low storage overhead.

VII. CONCLUSION

In this paper, we present a dynamic scheme to perform cache bypassing specifically for GPUs without the need of offline analysis. This paper first identifies GPU cache line reuse patterns and then introduces Ctrl-C to mitigate the data cache inefficiency problem in modern GPUs. Our evaluation results show that Ctrl-C is able to significantly reduce the MPKI and interconnect bandwidth demand. With Ctrl-C, cache sensitive applications can achieve a 1.42x speedup.

ACKNOWLEDGMENT

The authors would like to thank Dr. Amrit Panda and the anonymous reviewers for their insightful feedback. This work is supported in part by the National Science Foundation (Grant #CCF-1618039) and by Science Foundation Arizona under the Bisgrove Early Career Scholarship. The opinions, findings and conclusions or recommendations expressed in this manuscript are those of the authors and do not necessarily reflect the views of the Science Foundation Arizona.

REFERENCES