DaCache: Memory Divergence-Aware GPU Cache Management

Bin Wang[†], Weikuan Yu[†], Xian-He Sun[‡], Xinning Wang[†]

[†]Department of Computer Science
Auburn University
Auburn, AL, 36849
{bwang,wkyu,xzw0033}@auburn.edu

[‡]Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
sun@iit.edu

ABSTRACT

The lock-step execution model of GPU requires a warp to have the data blocks for all its threads before execution. However, there is a lack of salient cache mechanisms that can recognize the need of managing GPU cache blocks at the warp level for increasing the number of warps ready for execution. In addition, warp scheduling is very important for GPU-specific cache management to reduce both intra- and inter-warp conflicts and maximize data locality. In this paper, we propose a Divergence-Aware Cache (*DaCache*) management that can orchestrate L1D cache management and warp scheduling together for GPGPUs. In DaCache, the insertion position of an incoming data block depends on the fetching warp's scheduling priority. Blocks of warps with lower priorities are inserted closer to the LRU position of the LRU-chain so that they have shorter lifetime in cache. This fine-grained insertion policy is extended to prioritize coherent loads over divergent loads so that coherent loads are less vulnerable to both inter- and intra-warp thrashing. DaCache also adopts a constrained replacement policy with L1D bypassing to sustain a good supply of Fully Cached Warps (FCW), along with a dynamic mechanism to adjust FCW during runtime. Our experiments demonstrate that DaCache achieves 40.4% performance improvement over the baseline GPU and outperforms two state-of-the-art thrashing-resistant techniques RRIP and DIP by 40% and 24.9%, respectively.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Design, Performance

Keywords

GPU; Caches; Memory Divergence; Warp Scheduling

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.

http://dx.doi.org/10.1145/2751205.2751239 .

1. INTRODUCTION

Graphics Processing Unit (GPU) has proven itself as a viable technology for a wide variety of applications to exploit its massive computing capability. It allows an application to be programmed as thousands of threads running the same code in a lock-step manner, in which warps of 32 threads can be scheduled for execution in every cycle with zero switching overhead. The massive parallelism from these Single-Instruction Multiple Data (SIMD) threads helps GPUs achieve a dramatic improvement in computational power compared to CPUs. To reduce the latency of memory operations, GPU has employed multiple levels of data caches to save off-chip memory bandwidth when there is locality within the accesses.

Due to massive multithreading, per-thread data cache capacity often diminishes. For example, Fermi supports a maximum of 48 warps (1536 threads) on each Streaming Multiprocessor (SM), and these warps share 16KB or 48KB L1 Data Cache (L1D) [27]. Thus coalescing each warp's per-thread global memory accesses into fewer memory transactions not only minimizes the consumption of memory bandwidth, but also alleviates cache contention. But when a warp's accesses cannot be coalesced into one or two cache blocks, which is referred to as memory divergence, its cache footprint is often boosted by one order of magnitude, e.g., from 1 to 32 cache blocks. This leads to severe contention among warps, i.e., inter-warp contention, on limited L1D capacity.

Under the lock-step execution model, a warp is not ready for execution until all of its threads are ready (e.g., no thread has outstanding memory request). Meanwhile, cache-sensitive GPGPU workloads often have high intra-warp locality [32, 33], which means data blocks are re-referenced by their fetching warps. Intra-warp locality is often associated with strided accesses [19, 35], which lead to divergent memory accesses when stride size is large. The execution model, intra-warp locality, and potential memory divergence together pose a great challenge for GPU cache management, i.e., data blocks fetched by a divergent load instruction should be cached as a wholistic group. Otherwise, a warp is not ready for issuance when its blocks are partially cached. This challenge demands a GPU-specific cache management that can resist inter-warp contention and minimize partial caching. Though there are many works on thrashing-resistant cache management for multicore systems [30, 10, 17, 21], they are all divergence-oblivious, i.e., they make caching decisions at the per-thread access level rather than at the per-warp instruction level.

Recently, GPU warp scheduling has been studied to alleviate inter-warp contention from its source. Several warp scheduling techniques have been proposed based on various heuristics. For example, CCWS [32], DAWS [33], and CBWT [5] rely on detected L1D locality loss, aggregated cache footprint, and varying on-chip network latencies, respectively, to throttle concurrency at runtime.

Limiting the number of actively scheduled warps directly reduces inter-warp contention and delivers higher reductions of cache misses than the Belady [2] replacement algorithm in highly cache-sensitive GPGPU benchmarks [32]. We observe that coherent loads may also carry high intra- and inter-warp locality, but are vulnerable to the thrashing from both inter- and intra-warp divergent loads. However, warp scheduling can only be exploited to alleviate inter-warp contention at a coarse granularity, i.e., warp level. Thus there is still a need of a salient cache mechanism that can manage L1D locality at both levels and, more importantly, sustain a good supply of Fully Cached Warps (FCW) to keep warp schedulers busy.

Taken together, for a greater good on reducing cache misses and maximizing the occupancy of GPU cores, it is imperative to integrate warp scheduling with the GPU-specific cache management for a combined scheme that can overcome the inefficiency of existing GPU caches. To this end, we present a Divergence-Aware Cache (DaCache) management scheme to mitigate the impacts of memory divergence on L1D locality preservation. Based on the observation that warp scheduling shapes the locality pattern inside L1D access stream, DaCache gauges insertion positions of incoming data blocks according to the fetching warp's scheduling priority. Specifically, new blocks are inserted into L1D in an orderly manner based on their issuing warps' scheduling priorities. DaCache also prioritizes coherent loads over divergent loads in insertion to alleviate intra-warp contention. In addition, cache ways are conceptually partitioned into two regions, locality region and thrashing region, and replacement candidates are constrained within thrashing region to increase thrashing resistance. If no replacement candidate is available in thrashing region, L1D bypassing is enabled. We propose a simple mechanism to dynamically adjust the partitioning. All these features in our DaCache design need simple modifications to existing LRU caches.

In summary, this paper makes the following contributions:

- We evaluate caching effectiveness of GPU data caches for both memory-coherent and memory-divergent GPGPU benchmarks, and present the problem of partial caching in existing GPU cache management.
- We propose a Divergence-Aware Cache management technique, namely DaCache, to orchestrate warp scheduling and cache management for GPGPUs. By taking prioritization logic of warp scheduling into cache management, thrashing traffic can be quickly removed so that cache blocks of the most prioritized warps can be fully cached in L1D; in turn the increased number of fully cached loads provides more ready warps for warp schedulers to execute.
- We design a dynamic partitioning algorithm in DaCache to increase thrashing resistance and implement it in a cycle-accurate simulator. Experimental results show that it can improve caching effectiveness and improve the performance by 40.4% over baseline GPU architecture, outperform two thrashing resistance cache management, RRIP and DIP, by 40% and 24.9%, respectively.

The rest of paper is organized as follows: Section 2 introduces the baseline GPU; Section 3 summarizes the major characteristics of the evaluated GPGPU benchmarks and our motivation of DaCache; Section 4 details the design of DaCache; Experimental results and related work are presented in Section 5 and Section 6, respectively. Section 7 concludes the paper.

2. BASELINE GPU ARCHITECTURE

In this work we study modifications to a Fermi-like baseline GPU architecture as shown in Figure 1. In each Streaming Multiprocessor, two hardware warp schedulers independently manage

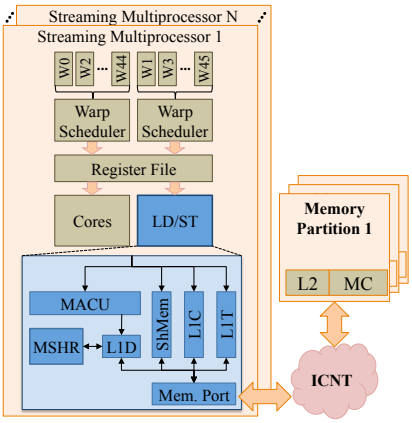


Figure 1: Baseline GPU Architecture.

all active warps. In each cycle, warp scheduler issues one warp among all the ready warps to execute in cores or Load/Store Unites (LD/ST) [24, 27, 28], depending on the warp's pending instruction. Once a memory instruction to global memory is issued, it's first sent to Memory Access Coalescing Unit (MACU) for access generation. MACU coalesces per-thread memory accesses to minimize off-chip memory traffic. For example, when 32 threads of a warp access 32 consecutive words in a cacheline-aligned data block, MACU will only generate one memory access to L1D. Otherwise, multiple memory accesses are generated to fetch all needed data. In the rest of this paper, the memory instructions that incur more than 2 uncoalescable memory accesses are called divergent instructions, while the others are called coherent instructions.

The resultant memory accesses from MACU are sequentially serviced by L1D. For a load access, if there is a cache hit, requested data are sent to register file; upon cache miss, if there is Missing Status Holding Register (MSHR) entry available, a request is generated and buffered into a queue in *Memory Port*. MSHR tracks in-flight requests and merges requests to the same missing data block. An MSHR entry is reclaimed when its corresponding memory request is back and all accesses to the block are serviced. Accesses missing in L1D are replayed when no MSHR entries are available. Cache lines are reserved for outstanding requests. Without coherence support for global data, L1D writes through dirty data and evicts cache lines on write hits. Buffered memory requests are sent to target memory partitions via an interconnect (*ICNT*). Each memory partition mainly consists of a L2 data cache and a memory controller (*MC*) that manages off-chip memory devices.

3. MOTIVATION

In this section, we evaluate GPU cache performance to understand application behaviors on a cache hierarchy similar to that in current GPUs. We use memory intensive benchmarks from Rodinia [4], PolyBench/GPU [11], SHOC [7], and MapReduce [14]. For each benchmark, Table 1 lists a brief description and the input size that we use for performance evaluation. Benchmark SC repetitively invokes the same kernel 290 times with the default input size (16K points) until the computation completes. Since the simulation is several orders of magnitude slower than real hardware, we only enable two kernel invocations in SC so that the simulation time is reasonable with larger input size (256K points). All of the other benchmarks, ranging from 70 million to 6.8 billion instructions, are run to completion. The benchmarks are categorized into memory-divergent and memory-coherent ones, depending on the dynamic divergence of load instructions in these benchmarks. In general, memory-divergent benchmarks are more sensitive to cache

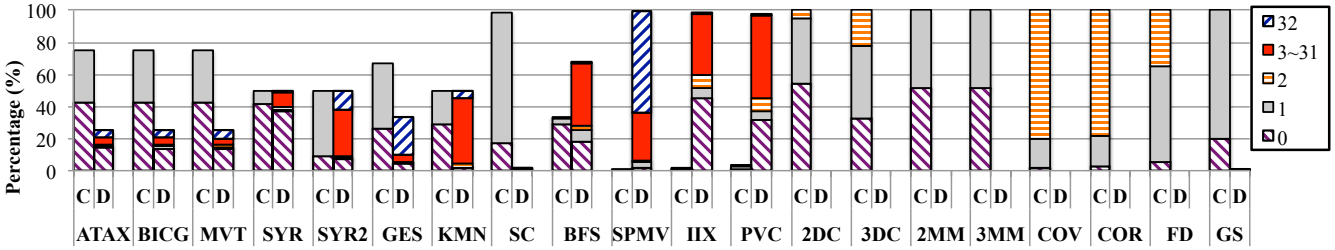


Figure 2: Distribution of Misses Per Load Instruction (MPLI) in L1 data cache. MPLIs are categorized into five groups: 0 (MPLI=0), 1 (MPLI=1), 2 (MPLI=2), 3∼31 (3⩽ MPLI ⩽ 31), and 32 (MPLI=32). MPLIs for coherent (*C*) and divergent (*D*) load instructions are accumulated separately. Each of the benchmarks on the right of the figure has only *C* bar for coherent instructions.

Table 1: GPGPU Benchmarks (CUDA)

#	Abbr.	Application	Suite	Input	Branch
Memory Divergent Benchmarks					
1	ATAX	matrix-transpose and vector mul.	[11]	$8K \times 8K$	N
2	BICG	kernel of BiCGStab linear solver	[11]	$8K \times 8K$	N
3	MVT	Matrix-vector-product transpose	[11]	8K	N
4	SYR	Symmetric rank-K operations	[11]	512×512	N
5	SYR2	Symmetric rank-2K operations	[11]	256×256	N
6	GES	Scalar-vector-matrix mul.	[11]	4K	N
7	KMN	Kmeans Clustering	[4]	28K 4x features	N
8	SC	Stream Cluster	[4]	256K points	N
9	BFS	Breadth-First-Search	[4]	5M edges	Y
10	SPMV	Sparse matrix mul.	[7]	default	Y
11	IIX	Inverted Index	[14]	6.8M	Y
12	PVC	Page View Count	[14]	100K	Y
Memory Coherent Benchmarks					
13	2DC	2D Convolution	[11]	default	N
14	3DC	3D Convolution	[11]	default	N
15	2MM	2 Matrix Multiply	[11]	default	N
16	3MM	3 Matrix Multiply	[11]	default	N
17	COV	Covariance Computation	[11]	default	N
18	COR	Correlation Computation	[11]	default	N
19	FD	2D Finite Difference Kernel	[11]	default	N
20	GS	Gram-Schmidt Process	[11]	default	N

capacity than memory-coherent benchmarks. Recent works [32, 33, 19, 35] report that high intra-warp L1D locality exists among these cache-sensitive workloads. In addition, BFS, SPMV, IIX, and PVC also have rich branch divergence.

3.1 Cache Misses from Divergent Accesses

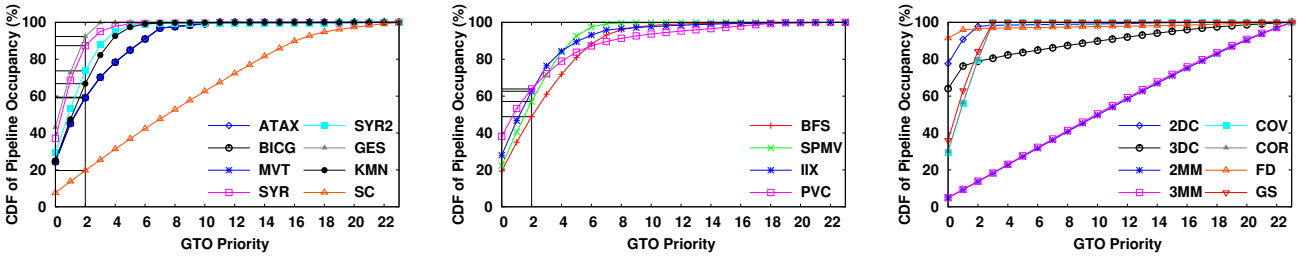
Within the lock-step execution model, a warp becomes ready when all of its demanded data is available; warps that have missing data, regardless of the data size, are excluded for execution. This execution model of GPU expects that all cache blocks of each divergent load instruction are cached as a unit when there is locality. However, conventional cache management is unaware of the GPU execution model and the collective nature of divergent memory blocks. As a result, some blocks of a divergent instruction can be evicted while others are still cached, resulting in a varying number of cache misses for individual loads. Metrics, such as Miss Rate and Misses Per Kilo Instructions (MPKI), are often used to evaluate the performance of cache management. In view of the wide variation of cache misses per instruction, we use **Misses Per Load Instruction (MPLI)** to quantify such misses in GPU L1D. Divergent load instructions that have misses in the range from 1 to $\{Req(pc, w) - 1\}$ are considered as being partially cached, where $Req(pc, w)$ is the number of cache accesses that warp w incurs at memory instruction pc. If a load instruction has no cache miss, it's considered as being fully cached. MPLI can be calculated by counting the number of cache misses a load instruction experiences after all of its memory accesses are serviced by L1D.

Figure 2 shows the distribution of MPLIs across the 20 GPGPU benchmarks we have evaluated in this paper. For simplicity, MPLIs are categorized into five groups. For divergent loads, the two categories of *2* (MPLI=2) and *3∼31* (3⩽ MPLI ⩽ 31) in the figure together describe the existence of partial caching. Note that this range can only provide a close approximation for partial caching because branch divergence can reduce the number of uncoalescable memory accesses a divergent load can generate. For example, a warp with 16 threads can maximally generate 16 memory accesses for a divergent load, and an MPLI of 16 indicates full caching for this load of the warp. As we can see from the figure, coherent loads of the memory-divergent benchmarks do not experience the problem of partial caching because they all generate one memory access per instruction. However, divergent load instructions in these benchmarks greatly suffer from partial caching. Substantial amount of divergent loads in SYR2, KMN, BFS, SPMV, IIX, and PVC are partially cached. Memory-coherent benchmarks, such as 2DC, 3DC, COV, COR, and FD, also experience partial caching (MPLI=1), because their load instructions generate two memory accesses each time. Besides some cold misses and capacity misses, such prevalent cache misses due to partial caching can be caused by severe cache contention, resulting in early evictions of cache blocks after being used only once.

3.2 Warp scheduling and Cache Contention

In view of the severe cache misses as discussed in Section 3.1, we have further examined the impact of warp scheduling on L1D contention. GPU warp scheduling is often driven by a prioritization scheme. For example, in the baseline Greedy-Then-Oldest (GTO) warp scheduling, warps are dynamically prioritized by their "ages" and the oldest warps are preferentially prioritized at runtime. In order to quantify the cache contention due to aggressive warp scheduling, we measure the occupancy of warp schedulers by all active warps. Figure 3 shows the Cumulative Distribution Function (CDF) of warp scheduler occupancy when the evaluated benchmarks are scheduled under GTO prioritization. Typically these benchmarks have one fully divergent load (resulting in 32 accesses) and one coherent load (resulting in one access) in the kernel, so the cache footprint of each warp is 33 cache lines at runtime. Our baseline L1D (32 KB, 256 lines) can fully cache three warps for each warp scheduler. This means that L1D will inevitably be thrashed if the warps with GTO priorities lower than 3 are scheduled. For memory-divergent benchmarks in Figure 3(a), 58%∼91% of the total cycles are occupied by the top 3 prioritized warps. Since branch divergence reduces the number of accesses a divergent load can generate, as shown in Figure 3(b), the occupancy drops to 48%∼63% among benchmarks with both memory- and branch-divergence. Such variation in warp scheduling incurs immediate cache conflicts.

We categorize conflict misses into intra- and inter-warp misses [19]. An intra-warp miss refers to the case where a thread's data is evicted by other threads within the same warp (*misses-iwarp*); otherwise a conflict miss is referred to as inter-warp miss (*misses-xwarp*).



(a) Memory-Divergent (b) Memory- and Branch-Divergent (c) Memory-Coherent

Figure 3: The CDF of warp scheduler occupancy by all active warps. The percentage reflects the frequency that each warp is scheduled. *GTO priority* refers to the "age" of each warp. Since each of the two warp schedulers in an SM manages 24 warps, 0 represents the highest priority, while 23 represents the lowest priority. Our baseline L1D can typically accommodate three divergent warps for each warp scheduler.

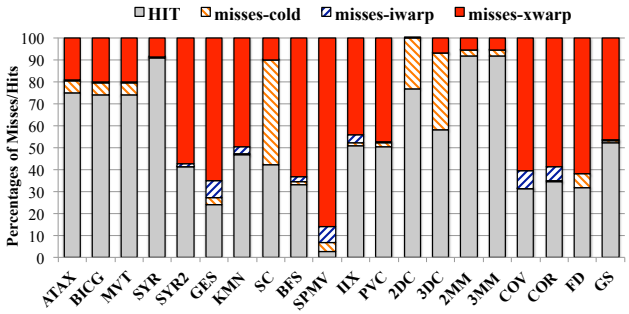


Figure 4: Categorization of L1D thrashing.

Meanwhile, we also present the percentages of cache hits (*HIT*) and cold misses (*misses-cold*). Figure 4 shows that the majority of cache misses are due to inter-warp conflicts, which in turn cause high MPLI as shown in Figure 2 and varied occupancy of warp schedulers as shown in Figure 3.

4. DIVERGENCE-AWARE GPU CACHE MANAGEMENT

As described in the previous section, divergent load instructions lead to many cache misses in L1D, especially inter-warp conflict misses. With more data blocks not being found in L1D, the number of warps that can be actively scheduled are significantly reduced. To address this problem, we propose *Divergence-Aware Cache (DaCache)* management for GPU. Based on the observation that the re-reference interval of cache blocks are shaped by warp schedulers, DaCache aims to exploit the prioritization information of warp scheduling, protect the cache blocks of highly prioritized warps from conflict-triggered eviction and maximize their chance of staying in L1D. In doing so, DaCache can alleviate the conflict misses across warps such that more warps can locate all data blocks from L1D for their load instructions. We refer to such warps as Fully Cached Warps (FCWs).

4.1 High-level Description of DaCache

Figure 5 shows a conceptual idea of DaCache in maximizing the number of FCWs. In this example, we assume four warps concurrently execute a for-loop body that has one divergent load instruction. At runtime, each warp generates four cache accesses in each loop iteration, and the fetched cache blocks are re-referenced across iterations. This is a common strided access pattern in our evaluated CUDA benchmarks. Ideally, all loads can hit in L1D due to high intra-warp locality. But severe cache contention caused by massive parallelism and scarce L1D capacity can easily thrash the locality in L1D. In order to resist thrashing, a divergence-oblivious cache management may fairly treat accesses from all warps, leading to the scenario that all warps miss one block in current iteration.

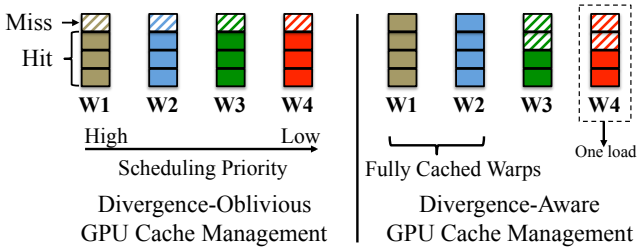


Figure 5: A conceptual example showing DaCache in maximizing the number of Fully Cache Warps.

By taking warp scheduling prioritization and memory divergence into consideration, DaCache aims at cache misses concentrated at warps that have lower scheduling priorities, such as *W3* and *W4*. Consequently, warps with higher scheduling priorities, such as *W1* and *W2*, can be fully cached so that they are immediately ready to execute the next iteration of the for-loop body.

DaCache relies on both warp scheduling-awareness and memory divergence-awareness to maximize the number of FCWs. This necessitates several innovative changes on GPU cache management policies. In general, cache management consists of three components: replacement, insertion, and promotion policies [38]. *Replacement policy* decides which block in a set should be evicted upon a conflicting cache access, *insertion policy* defines a new block's replacement priority, and *promotion policy* determines how to update the replacement priority of a re-referenced block. For example, in LRU caches, blocks at the LRU position are immediate replacing candidates; new blocks are inserted into the MRU position of the LRU chain; re-referenced blocks are promoted to the MRU position.

4.2 Gauged Insertion

In conventional LRU caches, since the replacement candidates are always selected from the LRU ends, blocks in the LRU-chains have different lifetime to stay in cache. For example, blocks at the MRU ends have the longest lifetime, while blocks at LRU ends have shortest lifetime. Based on this characteristic, locality of L1D blocks can be differentially preserved by inserting blocks at different positions in the LRU-chains according to their re-reference intervals. For example, blocks can be inserted into MRU, central, and LRU positions if they will be re-referenced in the immediate, near, and distant future, respectively. However, it is challenging for GPU caches to predict re-reference intervals of individual cache blocks from the thrashing-prone cache access streams.

Since there is often high intra-warp data locality among memory-divergent GPGPU benchmarks, the cache blocks of frequently scheduled warps have short re-reference intervals, while the blocks of infrequently warps have long re-reference intervals. Under GTO warp scheduling, old warps are prioritized over young warps and

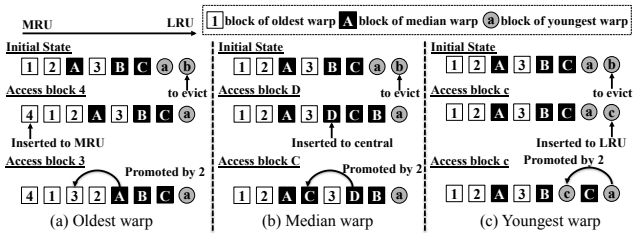


Figure 6: Illustrative example of insertion and promotion policies of DaCache.

thus are more frequently scheduled. Thus we can use each warp's GTO scheduling priority to predict its blocks' reference intervals. Based on this observation, the insertion position (*way*) in DaCache is gauged as:

$$way = min\{W_{Prio} \times N_{Sched} \times Width/N_{Set}, Asso - 1\} \quad (1)$$

where W_{Prio} is the issuing warp's scheduling priority, N_{Sched} is the number of warp schedulers in each SM, N_{Set} is the number of cache sets in L1D, $Width$ is the SIMD width, and $Asso$ is the cache associativity. Behind this gauged insertion policy, we assume the accesses from divergent loads (up-to $Width$ accesses) are equally distributed into N_{set} sets, and $Width/N_{Set}$ quantifies average intra-warp concentration in each cache set. Since L1D is shared by N_{Sched} warp schedulers, warps with the same priority but from different warp schedulers are assigned with the same insertion positions. Thus the cache blocks of consecutive warps from the same warp scheduler are dispersed by $N_{Sched} \times Width/N_{Set}$. For example, in our baseline GPU (2 warp schedulers per SM; 32 threads per warp; 32 sets per L1D), two warps with priorities of 0 and 2 are assigned insertion positions of 0 and 4, respectively. The gauged insertion policy is illustrated in Figure 6. In the figure, data blocks of "oldest warp", "median warp", and "youngest warp" are initially inserted into the MRU, central, and LRU positions, respectively. At runtime, the majority of the active warps are infrequently scheduled and share the LRU insertion position. By doing so, blocks are inserted in the LRU-chain in an orderly manner based on their issuing warps' scheduling priorities.

GPU programs often have a mix of coherent and divergent loads, which are assigned with the same insertion positions under the gauged insertion policy. Consequently, coherent loads will be interleaved with divergent loads. But interleaved insertion can make coherent loads vulnerable to thrashing from the bursty behaviors of divergent loads. The thrashing to coherent loads may not be limited to inter-warp contention. Figure 4 demonstrates the existence of intra-warp conflict misses in conventional LRU caches. We propose to explicitly prioritize coherent loads over divergent loads by inserting blocks of coherent loads into MRU positions, regardless of their issuing warps' scheduling priorities. But coherent loads may not carry any locality, and inserting their blocks into MRU positions is adversary to locality preservation. We use a victim cache to detect whether coherent loads have intra-warp locality, and then MRU insertion and LRU insertion are applied to coherent loads with and without locality, respectively. Motivated by the observation from Figure 3(b), we empirically use MRU insertion for divergent load instructions with no more than 5 memory requests.

Each entry of the victim cache has two fields, PC and data block tag. For a 48bit virtual address space, maximally the PC field needs 45 bits and the tag field needs 41 bits. Since only the mostly prioritized warp is sampled at runtime to detect the locality information of coherent loads, a 16-entry victim cache is sufficient across the evaluated benchmarks, which incurs only 172B storage overhead on each SM. The dynamic locality information of each coherent load is stored in a structure named Coherent Load Profiler (CLP). CLP entries have two fields, PC field (45 bits) and one flag field (1 bit) to indicate locality information. A 32-entry CLP incurs 184B storage overhead. Note that, when a load instruction is issued into LD/ST, memory access coalescing in MACU and CLP lookup can be executed in parallel. Once the locality information of a coherent load is determined, victim cache can be bypassed to avoid repetitive detection. Such storage overhead can be eliminated by embedding the potential locality information into PTX instructions via compiler support. We leave this as our future work.

Note that the insertion policy only gives an initial data layout in L1D to approximate re-reference intervals. During the runtime, the initial data layout can be easily disturbed because re-referenced blocks are directly promoted to the MRU positions, regardless of their current positions in the LRU-chain. In other words, this MRU promotion can invert the intention of DaCache insertion policy. Partially motivated by the incremental promotion in PIPP [38] that promotes re-referenced block by 1 position along the LRU-chain, DaCache also adopts a fine-grained promotion policy to cooperate with the insertion policy. Figure 6 illustrates a promotion granularity of 2 positions. Our experiments in Section 5.6 show that a promotion granularity of 4 achieves the best performance for the benchmarks we have evaluated.

4.3 Constrained Replacement

In general, in LRU caches, the block on the LRU end is considered as the replacement candidate. However, as we model cache contention by allocating cache block on miss and reserving blocks for outstanding requests [1], the block at the LRU position may not be replaceable. Then a replaceable block that is the closest to the LRU position is selected. Thus the replacement decision is no longer constrained on the LRU end, and any block in the set may be a replacement candidate. Such unconstrained replacement position makes inter-warp cache conflicts very unpredictable.

To protect the intention of gauged insertion, we introduce a constrained replacement policy in DaCache so that only a few blocks close to the LRU end can be replaced. This constrained replacement conceptually partitions the cache ways into two portions, locality region and thrashing region. Then replacement can only be made inside the thrashing region. This partitioning (p) can be calculated as: $p = \frac{Asso \times F}{SIMD\_Width/N_{Set}} - 1$, where F is a tuning parameter in the range between 0 and 1. Denoting the MRU and LRU ends with the way indexes of 0 and *Asso-1*, respectively, the locality region is located in the range from the 0^{th} to the p^{th} way of a cache set, while the thrashing region occupies the other ways. We tune the value of F to have the optimal static partitioning p. Besides, all sets in each L1D are equally partitioned.

Given the gauged insertion policy, this logical partitioning of L1D accordingly divides all active warps into two groups, locality warps and thrashing warps. If a warp's scheduling priority is higher than $(p+1)/N_{Sched}$, it's a thrashing warp; else it is a locality warp. The cache blocks of locality warps are inserted into the locality region using the gauged insertion policy so that they can be less vulnerable to thrashing traffic. By doing so, locality warps have a better chance to be fully cached and immediately ready for re-scheduling. In order to cooperate with such a constrained replacement policy, divergent loads of thrashing warps are exclusively inserted into LRU positions so that they can not pollute existing cache blocks in L1D. Though the 3 oldest warps managed by each warp scheduler, are mostly scheduled as shown in Figure 3, i.e., p=5 in our baseline, our experiments in Section 5.4 show that maintaining 2 FCWs per warp scheduler (p=3) actually achieves the optimal performance with the extended insertion and unconstrained replacement policies.

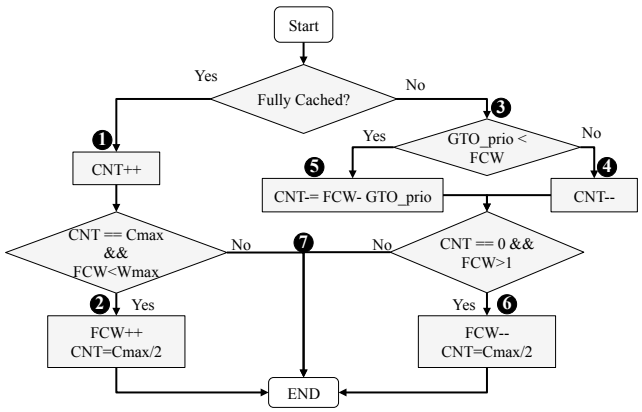


Figure 7: Flow of the proposed dynamic partitioning algorithm. *Fully Cached Warps (FCW)* is based on the number of fully cached loads (*CNT*) and each warp's GTO scheduling priority (GTO_prio).

With the constrained replacement policy, replacement candidates may not always be available. Thus we discuss two complementary approaches to enforce constrained replacement. The first approach is called **Constrained Replacement with L1D Stalling**. It's possible that a replacement candidate cannot be located within our baseline cache model, though at a very low frequency. Once this happens, the cache controller repetitively replays the missing access until one block in the thrashing region becomes replaceable. Stalling L1D is the default functionality within our cache model and then can be used with constrained replacement at no extra cost.

The second approach is called **Constrained Replacement with L1D Bypassing**. Instead of waiting for reallocating reserved cache blocks, bypassing L1D proactively forwards the thrashing traffic into lower memory hierarchy. Without touching L1D, bypassing can avoid not only L1D thrashing, but also memory pipeline stalls. When a bypassed request is back, its data is directly written to register file rather than a pre-allocated cache block [19]. In our baseline architecture, caching in L1D forces the size of missed memory requests to be cache block size. For each cache access of a divergent load instruction, only a small segment of the cache block are actually used, depending on the data size and access pattern. Without caching, the extra data in the cache block is a pure waste of memory bandwidth. Thus bypassed memory requests are further reduced to aligned 32B segments, which is the minimum coalesced segment size as discussed in [29].

4.4 Dynamic Partitioning of Warps

Our insertion and replacement policies rely on a static partitioning p, which incorporates the scheduling priorities of active warps into the cache management. However, the static choice of p is not very suitable in two important scenarios. Firstly, branch divergence reduces per-warp cache footprint so that the locality region is capable of accommodating more warps. It can be observed from Figure 3(b) that branch divergence enables more warps be actively scheduled. Secondly, kernels may have multiple divergent load instructions so that the capacity of locality region is only enough to cache one warp from each warp scheduler. For example, SYR2, GES, and SPMV have two divergent loads, while IIX and PVC have multiple divergent loads.

Thus we propose a mechanisms for dynamic partitioning of warps based on the accumulated statistics of fully cached divergent loads. Figure 7 shows the flow of dynamically adjusting *Fully Cached Warps (FCW)* based on the accumulated number of fully cached loads (*CNT*) and each warp's GTO scheduling priority (GTO_prio). At runtime, CNT is increased by 1 (❶) for each fully cached load.

Table 2: Baseline GPGPU-Sim Configuration

# of SMs	30 (15 clusters of 2)
SM Configuration	1400Mhz, Reg #: 32K, Shared Memory: 48KB, SIMD Width: 16, warp: 32 threads, max threads per SM: 1024
Caches / SM	Data: 32KB/128B-line/8-way, Constant: 8KB/64B-line/24-way, Texture: 12KB/128B-line/2-way
Branching Handling	PDOM based method [9]
Warp Scheduling	GTO
Interconnect	Butterfly, 1400Mhz, 32B channel width
L2 Unified Cache	768KB, 128B line, 16-way
Min. L2 Latency	120 cycles (compute core clock)
Cache Indexing	Pseudo-Random Hashing Function [26]
# Memory Partitions	6
# Memory Banks	16 per memory partition
Memory Controller	Out-of-Order (FR-FCFS), max request queue length: 32
GDDR5 Timing	$t_{CL} = 12, t_{RP} = 12, t_{RC} = 40, t_{RAS} = 28, t_{RCD} = 12, t_{RRD} = 6, t_{CDLR} = 5, t_{WR} = 12$

When CNT is saturated (CNT==Cmax), if FCW has not reached its maximum value (*Wmax*), FCW is increased by 1 and accordingly CNT is reset as *Cmax/2* to track fully cached divergent loads under the new partitioning (❷). For partially cached loads (❸), CNT is decreased differently depending on the issuing warp's scheduling priority. For instance, if a warp's scheduling priority is lower than FCW, CNT is decreased by 1 (❹); otherwise, CNT is decreased by FCW-GTO_prio (❺) to speed up the process of achieving the optimal FCW. When CNT reaches zero, FCW is decreased by 1 so that less warps are assigned into the locality region (❻). In our proposal, each warp scheduler has at least 1 warp in the locality region; while *Wmax* is equal to 48, which is the number of physical warps on each SM. Thus, in the corner cases when FCW is 1 or Wmax (❼), CNT will not be overflowed if it's saturated.

In order to implement the logic of dynamic partitioning, we first use one register (*Div-reg*) to mark whether a load is divergent or not, depending on the number of coalesced memory requests. *Div-reg* is populated when a new load instruction is serviced by L1D. We then use another register (*FCW-reg*) to track whether a load is fully cached or not. *FCW-reg* is reset when L1D starts to service a new load, and is set when a cache miss happens. When all the accesses of the load are serviced, *FCW-reg* being unset indicates a fully cached load. The logic of dynamic partitioning is triggered when a divergent load retires from the memory stage. We empirically use a 8-bit counter for CNT so that it can maximally record 256 consecutive occurrence of fully/partially cached loads, i.e., Cmax=256 in Figure 7. CNT is initialized as 128 while FCW is 4. This initial value of FCW is based on our experiments of static partitioning schemes showing that maintaining two FCWs for each warp scheduler has the best overall performance.

5. EXPERIMENTAL EVALUATION

We use GPGPU-Sim [1] (version 3.2.1), a cycle-accurate simulator, for the performance evaluation of DaCache. The main characteristics of our baseline GPU architecture are summarized in Table 2. The same baseline is also studied in [35, 36]. Jia et al. [19] reported that the default cache indexing method employed by this version of GPGPU-Sim can lead to severe intra-warp conflict misses, thus we use the indexing method from real Fermi GPUs, pseudo-random hashing function [26]. Actually this indexing method has been adopted in the latest versions of GPGPU-Sim. The following cache management techniques are evaluated:

LRU is the baseline cache management. Without further mentioning, all performance numbers are normalized to LRU.

DIP [30] consists of both LRU and MRU insertions. Cache misses are sampled from the sets that are dedicated to LRU and MRU insertions to determine a winning policy for all other

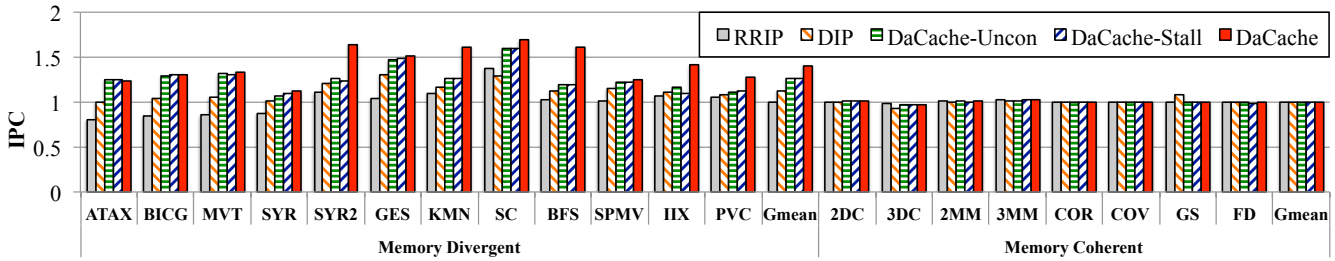
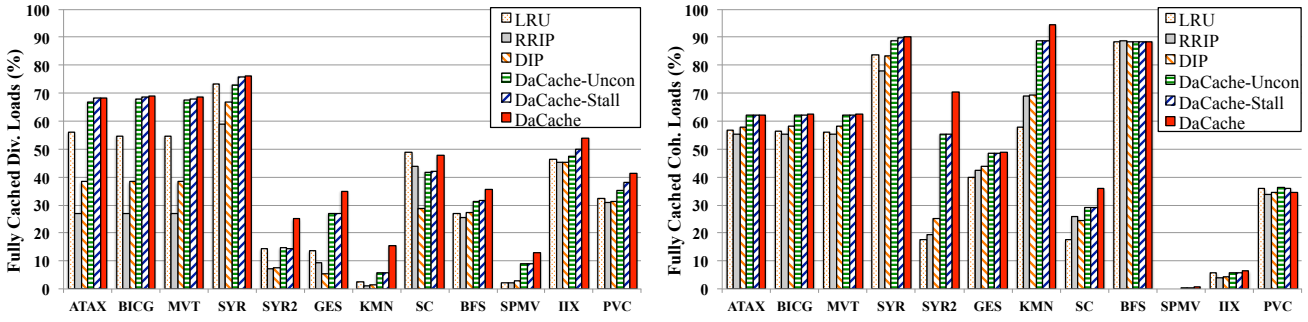


Figure 8: IPC of memory-divergent and memory-coherent benchmarks when various cache management techniques are used.



(a) Divergent Loads (b) Coherent Loads

Figure 9: Percentages of fully cached load instructions in memory divergent benchmarks.

"follower" sets, which is referred as set-dueling. In our evaluation, 4 sets are dedicated for each insertion policy and the other 24 sets are managed by the winning policy.

RRIP [17] uses Re-Reference Prediction Values (RRPV) for insertion and replacement. With an M-bit RRPV-chain, new blocks are predicted with RRPVs of 2^M-1 or 2^M-2, depending on the winning policy from the set-dueling mechanism. We implement RRIP with a Frequency Priority based promotion and a 3-bit RRPV chain.

DaCache consists of gauged insertion and incremental promotion (Section 4.2), constrained replacement with L1D bypassing (Section 4.3), and dynamic partitioning (Section 4.4). By default, DaCache has a promotion granularity of 4 and the locality region starts with hosting 2 warps from each warp scheduler. We evaluate DaCache variants with unconstrained replacement (*DaCache-Uncon*) and constrained replacement with L1D stalling (*DaCache-Stall*) to demonstrate the importance of using warp scheduling to guide cache management.

5.1 Instructions Per Cycle (IPC)

Figure 8 compares the performance of various cache management techniques for both memory-divergent and memory-coherent benchmarks. For memory-divergent benchmarks, RRIP on average has no IPC improvement. The performance gains of RRIP are balanced out by its loss in ATAX, BICG, MVT, and SYR, which exhibit LRU-friendly accesses patterns under GTO. Because of the intra-warp locality, highly prioritized warps leave large amount of blocks in the locality region that no other warps will re-reference, i.e., dead blocks, after they retire from LD/ST units. RRIP's asymmetric processes of promotion and replacement make it slow to eliminate the dead blocks, leading to inferior performance in these LRU-friendly benchmarks. Dynamically adjusting between LRU and MRU insertions makes DIP capable of both LRU-friendly and thrashing-prone patterns, thus DIP has 12.4% IPC improvement. In contrast, *DaCache-Uncon*, *DaCache-Stall*, and *DaCache* have an improvement of 25.9%, 25.6%, and 40.4%, respectively. The performance advantage of *DaCache-Uncon* proves the effectiveness of incorporating warp scheduling into L1D cache management. Based on this warp scheduling-awareness, constrained replacement with L1D stalling (*DaCache-Stall*) has no any extra performance gain. However, enabling constrained replacement with L1D bypassing achieves another improvement of 14.5% in *DaCache*.

Among the memory-coherent benchmarks, DIP has 8% performance improvement in GS. This is because GS has inter-kernel data locality, and inserting new blocks into LRU position when detected locality is low can help to carry data locality across kernels. We believe this performance improvement will diminish when data size is large enough. For the others, all of the cache management techniques have negligible performance impact. By focusing on memory divergence, DaCache and its variants have no detrimental impacts on memory coherent workloads. We believe DaCache is applicable to a large variety of GPGPU workloads.

5.2 Fully Cached Loads

The percentages of fully cached loads (Figure 9) explain the performance impacts of various cache management techniques on these memory-divergent benchmarks. As shown in Figure 9(a), LRU outperforms DIP and RRIP in fully caching divergent loads. Since GTO warp scheduling essentially generates LRU-friendly cache access patterns, LRU cache matches the inherent pattern so that the blocks of divergent loads are inserted into the contiguous positions of the LRU-chain. In contrast, DIP and RRIP dynamically insert blocks of the same load into different positions of LRU-chain and RRPV-chain, respectively, making it hard to fully cache divergent loads. Thus the performance impacts of RRIP and DIP mainly come from their capabilities in preserving coherent loads. As shown in Figure 9(b), for ATAX, BICG, MVT, and SYR, RRIP also achieves less fully cached coherent loads than LRU, thus it has worse performance than LRU in the four benchmarks; DIP recovers more coherent loads than LRU, but these gains are offset by loss in caching divergent loads, leading to marginal performance improvement. For SYR2, GES, KMN, SC, and BFS, RRIP and DIP improve the effectiveness of caching coherent loads, leading to the performance improvement in the five benchmarks.

DaCache-Uncon, *DaCache-Stall*, and *DaCache* constantly outperform LRU, RRIP, and DIP in fully caching loads, except for benchmark SC. This advantage comes from the following three

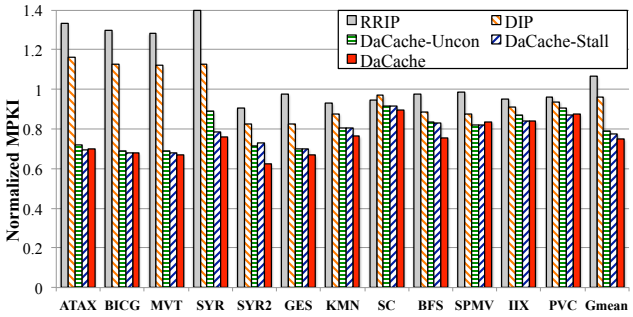


Figure 10: MPKI of various cache management techniques.

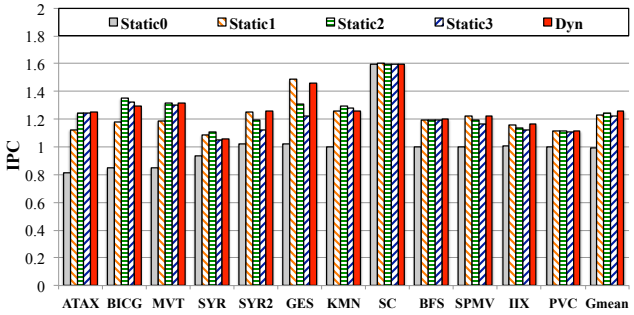


Figure 11: DaCache under static and dynamic partitioning.

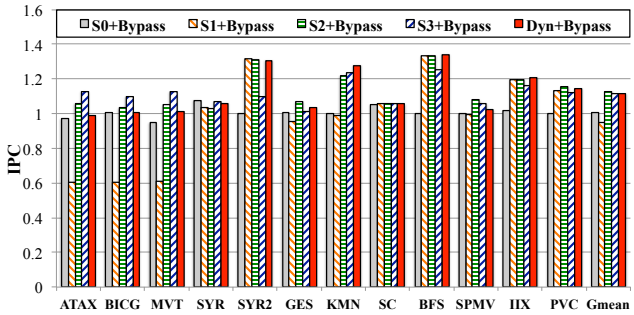


Figure 12: The impacts of using bypass to complement replacement policy under static and dynamic partitioning. Results are normalized to corresponding partitioning schemes.

factors. Firstly, guided by the warp scheduling prioritization, the gauged insertion implicitly enforces LRU-friendliness. Thus *DaCache-Uncon* achieves 35.1% more fully cached divergent loads. Secondly, deliberately prioritizing coherent loads over divergent loads alleviates the inter- and intra-warp thrashing from divergent loads. Thus *DaCache-Uncon* achieves 27.3% more fully cached coherent loads. Thirdly, constrained replacement can effectively improve the caching efficiency for highly prioritized warps. Based on *DaCache-Uncon*, constrained replacement with L1D stalling (*DaCache-Stall*) achieves 37.2% and 27.6% more fully cached divergent and coherent loads than LRU, respectively; while constrained replacement with L1D bypassing (*DaCache*) achieves 70% and 34.1% more fully cached divergent and coherent loads than LRU, respectively. In SC, the divergent loads come from the references to arrays of structs outside of a loop, and references to different members of the struct entry are sequential so that the LRU has the highest percentage of fully cached divergent loads (48.7%). But divergent loads in SC make up only a small portion of the total loads, therefore the number of fully cached coherent loads dominates the performance impacts.

5.3 Misses per Kilo Instructions (MPKI)

We also use MPKI to analyze the performance impacts of various cache management techniques on these memory-divergent benchmarks. As shown in Figure 10, except ATAX, BICG, MVT, and SYR, all of the five techniques are effective in reducing MPKIs. Because GPUs are throughput-oriented and rely on the number of fully cached warps to overlap long latency memory accesses, the significant MPKI increase of DIP in the four benchmarks is tolerated so that it doesn't have negative performance impacts. However, RRIP incurs on average a 32.5% increase in MPKIs in the four benchmarks, which leads to 14.5% performance degradation. Across the 12 benchmarks, on average, RRIP increases MPKIs by 6.4%, while DIP reduces MPKIs by 3.8%.

Meanwhile, *DaCache-Uncon*, *DaCache-Stall*, and *DaCache* consistently achieve MPKI reductions. On average, they reduce MPKIs by 20.8%, 22.4%, and 25%, respectively. Though *DaCache-Stall* reduces 1.6% more MPKIs than *DaCache-Uncon*, its potential performance advantage is compromised by adversely inserted L1D stall cycles. On the contrary, bypassing L1D in *DaCache* not only prevents L1D locality from being thrashed by warps with low scheduling priorities, but also enables these thrashing warps to directly access data cached in lower cache hierarchy. So 4.2% more MPKI reductions of *DaCache* brings 40.4% IPC improvement.

5.4 Static vs Dynamic Partitioning

Figure 11 examines the performance of DaCache when various static partitioning schemes and dynamic partitioning are enabled. For this experiment, the constrained replacement is disabled. *StaticN* means that *N* warps are cached in locality region. For example, in *Static0*, all blocks fetched by divergent loads are initially inserted into the LRU positions. Note that our baseline L1D is 8-way associative, Static3 and Static4 actually lead to identical insertion positions for all warps. Thus we only compare dynamic partition (*Dyn*) with *Static0*, *Static1*, *Static2*, and *Static3*.

Without any information from warp scheduling, *Static0* blindly inserts all blocks of divergent loads into LRU positions, thus it becomes impossible to predict which warps' cache block are more likely to be thrashed. On average, this inefficiency of *Static0* incurs 0.1% performance loss. On the contrary, by implicitly protecting 1, 2, and 3 warps for each warp scheduler, *Static1*, *Static2*, and *Static3* achieve performance improvement of 23%, 24.7%, and 21.9%, respectively. Note that *Static2* equally partitions L1D capacity into locality and thrashing regions, and the locality region is sufficient to cache two warps from each warp scheduler. Except IIX and PVC, all other benchmarks have maximally two divergent loads in each kernel, thus *Static2* has the best performance improvement. Our dynamic partitioning scheme (*Dyn*) achieves a performance improvement of 25.9%, outperforming all static partitioning schemes among the evaluated benchmarks. We expect this dynamic partitioning scheme can adapt to other L1D configurations and future GPGPU benchmarks that have diverse branch and memory divergence.

5.5 Constrained Replacement

Figure 12 explains when bypassing L1D can be an effective complement to replacement policy under static and dynamic partitioning. *SN* is equivalent to *StaticN* in Figure 11. The results are normalized to respective partitioning configurations. On average, constrained replacement with L1D bypassing incurs 0.6%, -5%, 12.8%, 11.4% and 11.6% performance improvement in *S0+Bypass*, *S1+Bypass*, *S2+Bypass*, *S3+Bypass*, and *Dyn+Bypass*, respectively. Note that these numbers are relative to partition-only configuration and are mainly used to quantify whether bypassing L1D is a viable complement to replacement policy. The performance degradation of *S1+Bypass* are mainly caused by ATAX, BICG, and MVT.

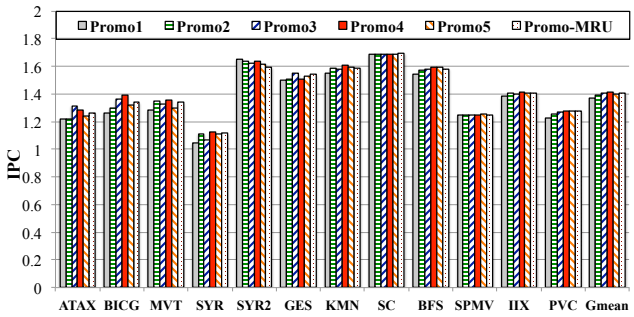


Figure 13: The impacts of promotion granularity under dynamic partitioning. *PromoN* means re-referenced blocks are promoted by *N* positions along the LRU-chain.

We observe that the three benchmarks have a large amount of dead blocks in L1D. Aggressive bypassing slows down the removal of dead blocks so that cache capacity is underutilized. We also analyzed the impact of stalling L1D as a complement to replacement policy under static and dynamic partitioning. We only observe negligible performance impacts, thus its results are not presented here due to the space limit.

5.6 Sensitivity to Promotion Granularity

Figure 13 analyzes the sensitivity of DaCache to the promotion granularity. In this experiment, *Promo-MRU* immediately promotes re-referenced blocks to the MRU positions, while *Promo1*, *Promo2*, *Promo3*, *Promo4*, and *Promo5* promote re-referenced blocks by 1, 2, 3, 4, and 5 positions respectively along the LRU-chain unless they reach the MRU position. As we can see, the majority of the benchmarks are sensitive to promotion granularity. These dead blocks are gradually demoted into thrashing region by inserting new blocks and/or promoting re-referenced blocks into locality region. Thus promotion granularity plays a critical role in eliminating dead blocks. Compared with LRU caches that directly promote re-referenced block to the MRU position, incremental promotion slowly promotes "hot" blocks towards the MRU position. The performance gap between *Promo1* (37.1%) and *Promo4* (41.6%) shows the importance of the promotion policy in DaCache.

6. RELATED WORK

There has been a large body of proposals on cache partitioning [12, 15, 16, 38, 3] and replacement policies [13, 31, 34] to increase the cache performance in CPU systems. However, these proposals do not handle the memory divergence issue within the massive parallelism of GPUs. Thus we mainly review the latest work within the context of GPU cache management.

6.1 Cache Management for GPU Architecture

L1D bypassing has been adopted by multiple proposals to improve the efficiency of GPU caches. Jia *et al.* [19] observed that certain GPGPU access patterns experience significant intra-warp conflict misses due to the pathological behaviors in conventional cache indexing methods, and thus proposed a hardware structure called Memory Request Prioritization Buffer (MRPB). MRPB reactively bypasses L1D accesses that are stalled by cache associativity conflicts. Chen *et al.* [6] used extensions in L2 cache tag to track locality loss in L1D. If a block is requested twice by the same SM, it's assumed that severe contention happens in L1D so that replacement is temporarily locked down and new requests are bypassed into L2. Chen *et al.* proposed another adaptive cache management policy, Coordinated Bypassing and Warp Throttling (CBWT) [5]. CBWT uses protection distance prediction [8] to dynamically assign each new block a protection distance (PD), which guarantees that the block will not be evicted if its PD has not reached zero. When no unprotected lines are available, bypassing is triggered and the PD values are decreased. CBWT further throttles concurrency to prevent NOC from being congested by aggressive bypassing. Different from the above three techniques, bypassing L1D in DaCache is coordinated with warp scheduling logic and a finer-grained scheme to alleviate both inter- and intra-warp contention. At runtime, bypassing is limited to the thrashing region which caches divergent loads from warps with low scheduling priorities and coherent loads with no locality.

Compiler directed bypassing techniques have been investigated to improve GPU cache performance in [18, 37], but the static bypassing decisions mainly work for regular workloads. DaCache is a hardware solution for GPU cache management and can adapt to program behavior changes at runtime. In some heterogeneous multicore processors, CPU and GPU cores share the Last Level Cache (LLC). There are also some work on cache management for this kind of heterogeneous systems [22, 25]. Although DaCache is designed for discrete GPGPUs, the idea of coordinating warp scheduling and cache management is also applicable to hybrid CPU-GPU systems.

Dong proposed an AgeLRU algorithm [23] for GPU cache management. AgeLRU uses extra fields in cache tags to track each cache line's predicted reuse distance, reuse count, and the active warp ID of the warp fetching the block, which together are used to calculate a score for replacement. The calculated score is reciprocal to each warp's age, i.e., older warps have higher scores to be protected. At runtime, the block with the lowest score is selected as replacement candidate and bypassing can be enabled when the score of the replacement victim is above a given threshold. By doing so, AgeLRU achieves the goal of preventing young warps from evicting blocks of old warps. DaCache doesn't need either storage in tag array or complicated calculation to assist replacement. By renovating the management policies, DaCache is more complexity-effective than AgeLRU to realize the same goal.

6.2 Warp Scheduling

There are several works that use warp scheduling algorithms to enable thrashing resistance in GPU data caches. Motivated by the observation that massive multithreading can increase contention in L1D for some highly cache-sensitive GPGPU benchmarks, Rogers et al. proposed a Cache Conscious Warp Scheduler (CCWS) [32] to limit the number of warps that issue load instructions when it detects loss of intra-warp locality. Following that, Rogers et al. also proposed a Divergence-Aware Warp Scheduling (DAWS) [33] to limit the number of actively scheduled warps whose aggregate cache footprint does not exceed L1D capacity. Besides, Kayiran et al. [20] proposed a dynamic Cooperative Thread Array (CTA) scheduling mechanism which throttles the number of CTAs on each core according to application characteristics. Typically, it reduces CTAs for memory-intensive applications to minimizing resource contention. By throttling concurrency, cache contention can be alleviated, and Rogers et al. reported in [32] that warp scheduling can be more effective than optimal cache replacement [2] in preserving L1D locality. However, throttling concurrency usually permits only a few warps to be active, though each warp scheduler is hosting a lot more warps that are ready for execution (maximally 24 warps in our baseline). Our work is orthogonal to these warp scheduling algorithms, because contention still exists in reduced concurrency. DaCache can be used to increase cache utilization under reduced concurrency and also uplift the resultant concurrency.

7. CONCLUSION

GPUs are throughput-oriented processors that depend on massive multithreading to tolerate long latency memory accesses. The latest GPUs all are equipped with on-chip data caches to reduce the latency of memory accesses and save the bandwidth of NOC and off-chip memory modules. But these tiny data caches are vulnerable to thrashing from massive multithreading, especially when divergent load instructions generate long bursts of cache accesses. Meanwhile, the blocks of divergent loads exhibit high intra-warp locality and are expected to be atomically cached so that the issuing warp can fully hit in L1D in the next load issuance. However, GPU caches are not designed with enough awareness of either SIMD execution model or memory divergence.

In this work, we renovate the cache management policies to design a GPU-specific data cache, *DaCache*. This design starts with the observation that warp scheduling can essentially shape the locality pattern in cache access streams. Thus we incorporate the warp scheduling logic into insertion policy so that blocks are inserted into the LRU-chain according to their issuing warp's scheduling priority. Then we deliberately prioritize coherent loads over divergent loads. In order to enable the thrashing resistance, the cache ways are partitioned by desired warp concurrency into two regions, the locality region and the thrashing region, so that replacement is constrained within the thrashing region. When no replacement candidate is available in the thrashing region, incoming requests are bypassed. We also implement a dynamic partition scheme based on the caching effectiveness that is sampled at runtime. Experiments show that DaCache achieves 40.4% performance improvement over the baseline GPU and outperform two state-of-the-art thrashing resistant cache management techniques RRIP and DIP by 40% and 24.9%, respectively.

Acknowledgments

This work is funded in part by an Alabama Innovation Award, and by National Science Foundation awards 1059376, 1320016, 1340947 and 1432892. The authors are very thankful to anonymous reviewers for their invaluable feedback.

8. REFERENCES

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[3] J. Chang and G. S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS*, 2007.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[5] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. W. Hwu. Adaptive Cache Management for Energy-efficient GPU Computing. In *MICRO*, 2014.

[6] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W.-M. W. Hwu. Adaptive Cache Bypass and Insertion for Many-core Accelerators. In *MES*, 2014.

[7] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU*, 2010.

[8] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving Cache Management Policies Using Dynamic Reuse Distances. In *MICRO*, 2012.

[9] W. W. L. Fung, I. Sham, G. L. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*, 2007.

[10] H. Gao and C. Wilkerson. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing. In J. Emer, editor, *JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship*, Saint Malo, France, 2010.

[11] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Innovative Parallel Computing*, 2012.

[12] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *MICRO*, 2007.

[13] E. G. Hallnor and S. K. Reinhardt. *A fully associative software-managed cache design*, volume 28. ACM, 2000.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, 2008.

[15] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS*, 2004.

[16] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS*, 2007.

[17] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. S. Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *ISCA*, 2010.

[18] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *ICS*, 2012.

[19] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *HPCA*, 2014.

[20] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs. In *PACT*, 2013.

[21] S. M. Khan, Y. Tian, and D. A. Jimenez. Sampling Dead Block Prediction for Last-Level Caches. In *MICRO*, 2010.

[22] J. Lee and H. Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *HPCA*, 2012.

[23] D. Li. *Orchestrating Thread Scheduling and Cache Management to Improve Memory System Throughput in Throughput Processor*. PhD thesis, University of Texas at Austin, May 2014.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, Mar. 2008.

[25] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *PACT*, 2013.

[26] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *HPCA*, 2014.

[27] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[28] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012.

[29] NVIDIA. CUDA C Programming Guide, 2013.

[30] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, Jr., and J. S. Emer. Adaptive Insertion Policies for High Performance Caching. In *ISCA*, 2007.

[31] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *ISCA*, 2005.

[32] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.

[33] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware Warp Scheduling. In *MICRO*, 2013.

[34] R. Subramanian, Y. Smaragdakis, and G. H. Loh. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In *MICRO*, 2006.

[35] B. Wang, Z. Liu, X. Wang, and W. Yu. Eliminating Intra-Warp Conflict Misses in GPU. In *DATE*, 2015.

[36] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter. Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design. In *PACT*, 2013.

[37] X. Xie, Y. Liang, G. Sun, and D. Chen. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *ICCAD*, 2013.

[38] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *ISCA*, 2009.