Dynamically Linked MSHRs for Adaptive Miss Handling in GPUs

Yongbin Gu, Lizhong Chen
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR 97331
{guyo,chenliz}@oregonstate.edu

ABSTRACT

Supporting a large number of outstanding memory requests in miss handling architecture (MHA) is critical for throughput processors such as GPUs to achieve high memory level parallelism. Conventional MHA is static in sense that it provides a fixed number of MSHR entries to track primary misses, and a fixed number of slots within each entry to track secondary misses. This leads to severe entry or slot under-utilization and poor match to practical workloads, as the number of memory requests to different cache lines can vary significantly. In this paper, we propose *Dynamically Linked MSHR (DL-MSHR)*, a novel approach that dynamically forms MSHR entries from a pool of available slots. This approach can self-adapt to primary-miss-predominant applications by forming more entries with fewer slots, and self-adapt to secondary-miss-predominant applications by having fewer entries but more slots per entry. Evaluation results show that, compared with the conventional MSHRs, the proposed DL-MSHR is able to reduce reservation fails in MSHRs by 88.1%, improve MSHR utilization by 53.7% and increase the overall IPC of a wide range of workloads by 19.2%, on average, with only 0.6% and 0.1% area overhead on L1D and L2 cache, respectively.

1 INTRODUCTION

Many-core processors have an increasing demand for higher memory level parallelism (MLP) to achieve better performance [8]. Consequently, a large number of outstanding memory requests need to be tracked simultaneously in the memory subsystem by the miss handling architecture (MHA). This demand becomes more pressing in GPUs, as the single instruction multiple threads (SIMT) model can easily execute hundreds to thousands of threads concurrently, resulting in numerous memory requests pending in the memory hierarchy. Thus, it is imperative to design miss handling architectures that can process and track cache misses at a matching rate.

MHA has been evolving continuously in the past years, with most of today's GPUs having MHA based on *Miss Handling Status Registers (MSHRs)*. When a requested data is not found in the cache and sent to the next memory level, the associated MSHR tracks the cache miss by temporally storing the requester ID, cache block address, requested data tag, and other related information until the data is returned from the lower level. A typical MHA may have dozens of MSHR entries (e.g., 32 or 64) and each entry may in turn have multiple slots (e.g., 4 or 8). An entry is allocated to the primary miss to a cache line, and the slots within the entry are allocated to the secondary misses to the same cache line while the primary miss is pending. The MHA is critical to memory level parallelism, as no new memory requests can be processed if there is no free entry or slot available in the MHA.

While the above architecture works well to a certain degree, it may no longer be sufficient in handling the increasing diverse miss behaviors in GPU workloads. The main issue with the conventional array-based MSHRs is that the entire structure is static, in the sense that every entry has the same number of slots and this number is fixed after manufacturing. However, it is unlikely that every cache line has the same number of misses. While some entries are in high demand for slots, other entries may have multiple slots being unused. To understand the workload demand in practice better, we evaluated a number of applications from three widely used GPU benchmark suites. Results show that the cache misses in most benchmarks are predominant by either primary misses or secondary misses. This highlights that the entry/slot utilization in conventional MSHRs would be poor when running the common workloads, and that the structure would not perform well for all the applications due to the diverse miss behaviors. A direct and naive way to address this issue is to add more entries and slots. This method not only incurs substantial overhead (e.g., 22.3% overhead in terms of L2 cache area, as shown later), but also has limited effectiveness as certain applications may demand over 30 secondary misses to the same cache line (thus requiring 30 slots per entry) but only need 2 to 3 entries. It is simply impractical to increase MSHRs from the typical 4-8 slots per entry to that size. To address this important problem, innovative solutions are needed to utilize the MSHR resources smartly.

In this paper, we propose *Dynamically Linked MSHR (DL-MSHR)*, a novel approach that allocates miss handling resources flexibly and adaptively to meet the diverse miss behaviors of applications. In DL-MSHR, entries are formed dynamically from a pool of available slots. A slot can be assigned as an independent entry for processing a primary miss, or can be linked after another slot in an existing entry to increase the capacity of processing secondary misses. The number of slots that each entry has depends on the frequency of memory accesses to the corresponding cache line. This approach self-adapts to primary-miss-predominant applications by forming more entries with fewer slots, and adapts to secondary-miss-predominant applications by having fewer entries but more slots per entry. We also propose four additional optimization techniques to further increase the efficiency of DL-MSHR. Evaluation results show that, compared with conventional MSHRs, the proposed DL-MSHR is able to reduce the primary- and secondary-miss-induced reservation fails in MSHRs by 88.1%, improve the MSHR utilization by 53.7%, and

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06.
https://doi.org/10.1145/3330345.3330390

increase the overall IPC by 19.2% with only 0.6% and 0.1% area overhead on L1D and L2 cache, respectively. Moreover, DL-MSHR can complement existing techniques and achieve an additional 26.3% IPC improvement on top of MRPB (Memory Request Prioritization Buffers) [10]. The average IPC of DL-MSHR is even 8.0% higher than the conventional MSHR configured with 4X the amount of hardware, i.e., doubling the number of entries *and* doubling the number of slots per entry.

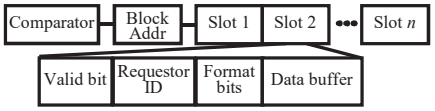
2 BACKGROUND

Miss handling architecture (MHA) is critical to memory level parallelism and system performance, as MHA feeds and tracks concurrent miss requests that are issued to the next level of memory hierarchy. Over the years, miss handling architecture has been evolving continuously and has unlocked an increasing amount of parallelism that can be achieved by cache and memory. This section explains several key considerations of MHA that lead to the explicitly-addressed, MSHR-based MHA design today.

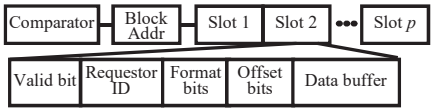
Lockup cache vs. lockup-free cache. When cache was originally introduced, the associated MHA can handle only one outstanding miss at a time (i.e., lockup cache). To read a data, the data address is used to search the cache. A cache hit returns the requested data right away; whereas a cache miss requires the MHA to first record the pertinent information of the request and then issue the request to the next level in the memory hierarchy. Before the data is back, the cache does not process new misses and is "locked up". Writing data is similar (as most cache designs use write-on-allocate policies), except that the MHA needs to provide a data buffer to store the new data temporarily before the corresponding cache line is allocated and available.

To support lockup-free caches, multiple *Miss Status Holding Registers* (MSHRs) are added to the MHA to keep track of multiple outstanding misses concurrently[14]. Each cache miss is allocated one MSHR entry which records the information of the miss, such as the requester ID, cache block address, requested data tag, new data for write-back (in case of writing), etc. Once the requested data is returned, the data can be forwarded back to the corresponding requester based on the information retrieved from the MSHR. The cache can accept new misses from processing cores as long as there are free MSHRs available.

Primary miss vs. secondary miss. As the smallest unit for data transferring between two cache levels is a cache line rather than individual words, additional optimization opportunity exists in combining multiple data requests to the same cache line. To exploit this opportunity, cache misses are divided into two types. A *primary* miss occurs when the cache line containing the requested word does not exist in the cache and a new MSHR entry needs to be allocated. A *secondary* miss occurs when the requested word shares the same cache line of an outstanding miss, in which case no new request needs to be issued to the next level since the requested cache line is already on the way. Note that the requested word in the secondary miss could be to a different word in the same cache line, or to the same word as the primary miss but from a different requester (e.g., a different core). To accommodate this, each MSHR entry further consists of several slots to keep track of individual word requests. An address comparator is associated with each MSHR entry to check if any incoming cache miss shares the same block address of the cache line that the MSHR is allocated to. if yes, a free slot in the matching MSHR entry is be needed; if not, a free MSHR entry would be needed. The comparison is done in parallel across all the MSHRs, similar to a fully associative cache. In this paper, we use the term *entry-full* to refer to the case where no free MSHR entry is available, and use *merge-full* to refer to the situation where no free slot is available within an MSHR entry.



(a) Implicitly addressed MSHR (*n*=cache line size/word size)



(b) Explicitly addressed MSHR (*p* can be smaller than *n*)

Figure 1: Implicitly and explicitly addressed MSHRs.

Implicitly vs. explicitly addressed MSHR. To realize the functionality of MSHRs in hardware, Kroft proposes an implementation based on implicitly addressed MSHRs [14]. As depicted in Figure 1a, in this architecture, an MSHR entry provides a pre-allocated slot for each addressable word in a cache line. All the slots share the same block address, but the offset bits within a block (to specify each word) do not need to be recorded in a slot (i.e., the words in a block are implicitly addressed by using the position of the slots). If a particular word in a cache line is requested, the requester ID and other related information is recorded in the corresponding slot. As each word in a cache line has exactly one slot, an MSHR entry is able to track multiple secondary misses, provided that they request different words in a cache line. However, secondary misses to the same outstanding word are denied because there is no place to store more than one copy of tracking information. Although this design has simple control, having at most one outstanding miss per word is a very severe limitation, especially in the many-core era. Moreover, reserving one slot per word may lead to very low efficiency of MSHR given the large cache line size in contemporary processors (e.g., 32 words).

To overcome the drawbacks of the above design, Farkas and Jouppi [6] propose the explicitly addressed MSHR. As illustrated in 1b, the number of slots, *p*, in an MSHR entry does not have to be the same as the number of words in a cache line (*p* is the same across all the MSHRs). Instead, every slot is generic and can be used to track any word in the cache line. Consequently, the offset of the word needs to be explicitly recorded in the slot. Although the offset requires additional bits, the achieved savings in the reduced slots and the benefits of increased flexibility far outweigh the overhead, which makes this design the *de facto* MHA in most of the current commercial processors including Core i7[4], Xeon E5[33], and GTX960[23].

```
// Kernel in BlackScholesGPU
1: __global__ void BlackScholesGPU(
2: float *d_CallResult, float *d_PutResult,
3: float *d_StockPrice, float *d_OptionStrike,
4: float *d_OptionYears, float Riskfree,
5: float Volatility, int optN 6: ) {
7: int tid = blockDim.x * blockIdx.x + threadIdx.x;
8: int THREAD_N = blockDim.x * gridDim.x;
9: for(int opt = tid; opt < optN; opt += THREAD_N)
10: BlackScholesBodyGPU(
11:       d_CallResult[opt],
12:       d_PutResult[opt],
13:       d_StockPrice[opt],
14:       d_OptionStrike[opt],
15:       d_OptionYears[opt],
16:       Riskfree,
17:       Volatility
18: ); }
```

Figure 2: Blackscholes (primary-miss-predominant).

```
// Kernel in Aligned Types
1: template<class TData>__global__ void testKernel(
2: TData *d_odata, TData *d_idata, int nE)
3: {
4: int tid = blockDim.x * blockIdx.x + threadIdx.x;
5: int numThreads = blockDim.x * gridDim.x;
6: for(int pos = tid; pos < nE; pos += numThreads)
7:       d_odata[pos] = d_idata[pos];
8: }
```

Figure 3: AlignedType (secondary-miss-predominant).

While the previous discussions mainly focus on L1 cache for primary and secondary misses, similar situations also exist in L2 cache, but at the granularity of cache lines (instead of words). A cache line in a shared L2 may be accessed by multiple private L1 caches in different cores, thus requiring a multi-slot L2 MSHR entry to track these requests. For example, a private L1 may request a cache line from L2. If the line is not present in the L2, an L2 MSHR entry is allocated to track this primary miss while the line is being fetched from the memory. Meanwhile, if another private L1 cache has a write request to L2 for the same line, this request is allocated another slot in the above entry (i.e., secondary miss), and the write data from the write request is temporarily stored in the data buffer of that slot [1]. The explicitly addressed MSHR design also works more efficiently than the implicit one for L2, as there is no need to provide a reserved slot for each L1. Note that, if any of the read and write request results in the replacement of a dirty line, the dirty line does not need a MSHR slot; instead, it is placed into the evicted buffer and later written back to the memory.

3 MOTIVATION

The success of the explicitly addressed MSHR design demonstrates the importance of having an efficient miss handling architecture for enabling memory-level parallelism. However, this architecture may no longer be sufficient in handling the increasing diverse application miss behaviors.

[1]There can be multiple read and write requests to the same cache line in L2. To ensure correctness (e.g., consider the sequence of W1, R1, W2, R2), the write data from different private L1 cache requests cannot be combined in an L2 MSHR entry, thus requiring each slot to have a data buffer.

3.1 Diverse Application Cache Miss Behaviors

We first characterize applications by examining whether their predominant misses are primary or secondary misses. The results can help to understand the diverse demands on miss handling architecture. While several works have studied GPU workloads in detail, to our knowledge, no research has examined from the perspective of cache miss types, as defined below.

Primary-Miss-Predominant Applications. This type of applications exhibit a high demand for MSHR entries but not the slots within an entry. As an example, Figure 2 shows the kernel of the *blackScholes* benchmark from the NVidia SDK [24] that is primary-miss-predominant. For this kernel, there are 7 different floating variables from line 11 to 17 (4 bytes each) that need to be loaded before further calculation. If running on a GTX750Ti using all the 640 CUDA cores simultaneously (5 streaming multiprocessors (SMs) × 128 cores/SM = 640), up to 140 cache lines (640 × 7 variables × 4 bytes / 128 byte/line = 140, assuming perfect coalescing) could be requested in a cycle which, theoretically, needs 140 MSHR entries to track the information. Hence, the 32 entries of MSHRs in GTX750Ti are very easy to cause execution stall. The primary-miss-predominant applications can significantly benefit from an increase in the number of MSHR entries.

Secondary-Miss-Predominant Applications. Applications in this category have a high demand for the slots in MSHR entries but less so for MSHR entries. Figure 3 shows the kernel of the *alignedTypes* benchmark involving array operations. Array elements are usually stored sequentially in the address space. When multiple threads are executing this kernel simultaneously, these threads may likely access the same cache lines with good spatial locality and high number of secondary misses. Assuming the floating type for the TData template in line 1, there can be up to 128 byte/line / 4 bytes = 32 requests, which greatly exceeds the 8 slots in each MSHR entry in GTX750Ti. To increase the capacity of handing secondary misses, more slots have to be added. In the current MSHR architecture, this is very costly as 1) each slot contains a data buffer for the possible write miss, and 2) every MSHR entry has the same number of slots, so even adding one slot per entry would considerably increase hardware overhead.

We studied a number of applications from the NVidia SDK [24], Rodinia [2, 3] and Parboil [29] benchmark suites. Figure 4 presents the breakdown of reservation fails resulted from primary misses and secondary misses (other sources of RFs account for less than 3%). GPGPU-Sim [1] with a typical configuration (more details in Section 5) is used to simulate the benchmarks. The figure exhibits a great diversity, with some applications such as *b+tree* and *bfs* having reservation fails predominately from secondary misses, and applications such as *blackSholes* and *scan* having reservation fails predominately from primary misses. These results indicates that a static, one-size-fits-all MSHR architecture may not be the most efficient design to handle diverse GPU workloads.

To verify the merge-full and entry-full phenomenons in MSHR are not synthetic issues of the simulator, we have developed a microbenchmark that tests the MSHR of a recent GPU. Our intention is not to expose the publicly unavailable MSHR details of real GPUs, but rather to show that merge-full and entry-full indeed create performance issues in recent GPUs. There are three kernels

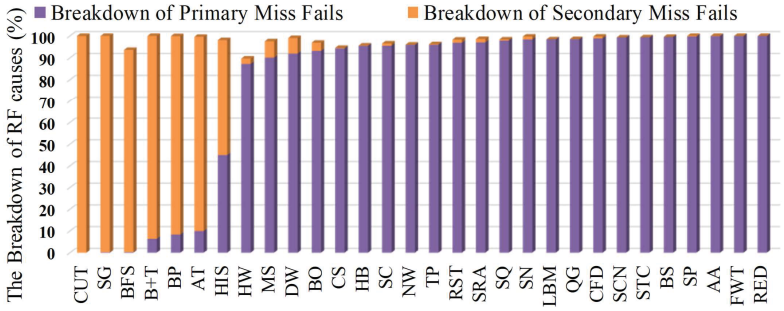


Figure 4: Breakdown of reservation fail (RF) causes.

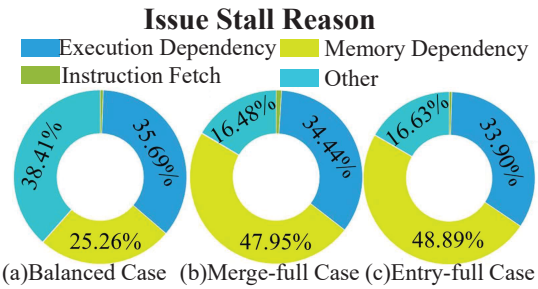


Figure 5: Percentage of execution stall reasons.

in this microbenchmark that represent a balanced case, a merge-full case, and a entry-full case, respectively. Each kernel consists of 64 blocks with 256 threads per block, totaling 16384 threads. The balanced case has relatively balanced primary and secondary misses to cache. In the merge-full case, half of the threads access the same cache line, which causes a large number of secondary misses. In the entry-full case, the threads occupy different cache lines (i.e. primary misses) as much as possible while minimizing secondary misses. A GTX960 graphics card is employed to execute the three kernels, and the NVidia Nsight tool [22] is used to collect the stalling data of the GPU. Figure 5 compares the percentage of various reasons that cause execution stall of the tested GPU. In the doughnut chart generated by Nsight, the percentage of stall from "memory dependency" increases from 25.3% in the balanced case to around 48% in the merge-full and entry-full cases, highlighting the severe negative impact of MSHR merge-full and entry-full behaviors. This is particularly evident in the merge-full case where half of the threads access the same cache line. One might expect that such access pattern would lead to a large number of cache hits and reduced data stall. However, the limited slots in each MSHR entry causes frequent merge-full situations and prevents further data requests from being serviced by the cache and MSHRs. While the performance impact of other benchmarks may not be as large as our microbenchmark, the experiment here demonstrates that the MSHR issue indeed exists in current practice.

3.2 Need for Dynamic Miss Handling

Under current MSHR architecture, addressing diverse application miss behaviors needs to increase the number of MSHR entries and slots. Note that both entries and slots need to be increased. Missing either of the two aspects would result in a class of applications to suffer from primary miss induced or secondary miss induced reservation fails. This approach is costly and inefficient for two major reasons. First, MSHRs are implemented as content-addressable memory (CAM). Each MSHR entry has an address comparator, and all the entries need to be searched in parallel up on a cache miss. This places a high capacitive load at the output gate of the upstream address decoders. Our evaluation based on CACTI 6.5 [19] confirms that the search delay and area cost of MHA rise superlinearly as the number of entries increases. However, these overhead is relatively small if the total number of entries is not large (i.e., the negative effect of superlinear growth becomes substantial *only* when the base number is large).

Second and more importantly, each slot contains a data buffer to temporarily store write-back data in case of a write miss. Thus, increasing the number of entries and/or slots would substantially increase the overall area of the MHA. For example, as shown in Section VI, doubling the number of entries and slots for L2 MSHR incur an area overhead of 22.3% in terms of L2 cache area, and 33.4% power overhead. Nevertheless, the performance gains from this are still very limited. Clearly, directly increasing the size of MSHR is not a cost-effective solution.

This calls for a flexible and dynamic MSHR design that can utilize hardware resources smartly. The opportunity comes from the fact that primary-miss-predominant applications need a large number of entries, but only few slots within each entry is occupied. Similarly, secondary-miss-predominant applications have high demand for the slots within certain entries, but many other MSHR entries (and their slots) may still be free. This opportunity is exploited in the approach proposed in this paper.

3.3 Other Related Work

GPU architecture has been improved from various aspects (e.g., [7, 11, 13, 15, 25, 30] and many others). However, only a few works have targeted the efficiency of miss handling architecture. To reduce cache look-up time and increase bandwidth, Tuck *et al.* [31] propose a hierarchical MHA, where a small MSHR file is provided at each cache bank to process the majority of secondary cache misses, and a large MSHR file that is shared by all the banks to handle long latency misses. In addition, Jahre *et al.* [9] propose to shrink the miss handling bandwidth for a specific core that delays the execution speed of other cores, thereby achieving a higher overall speedup. Neither of the above two designs can dynamically adjust the number of MSHR entries or slots that can be self-configured to best suit the needs of applications as proposed in this work. Loh[18] proposes Vector Bloom Filter (VBF) that can provide faster access for large MHA and can dynamically shrink MSHR capacity. However, VBF does not explore the opportunity in utilizing the unused slots in an entry that is already allocated to a primary miss, whereas DL-MSHR utilizes these slots by decomposing each entry into slots and dynamically linking them. In evaluation, we compare DL-MSHR with a perfect VBF where the MHA access time is one cycle regardless of the MHA size. In addition, Power *et al.* [26] propose a region-based coherence to reduce MSHR entries in the coherence directory in heterogeneous systems, and Qureshi *et al.* [27] propose a linked structure in V-Way cache to reduce the

unbalanced set access problem. Both works are related but have very different contexts than this work.

Also closely related to MHA are reservation fails, which may occur due to several reasons such as MSHR entry-full, MSHR merge-full, miss queue full, cache reservation full etc[12]. If the data request at the head of the request buffer to cache encounters a reservation fail, subsequent requests will be blocked even though they could have been processed in three cases: 1) the reservation fail is caused by entry-full with no available MSHR entry to track this primary miss, but subsequent requests could have been merged into other allocated MSHR entries (i.e., secondary misses); 2) the reservation fail is caused by merge-full with no slot to accept this secondary miss in a particular MSHR entry, but subsequent requests could have been assigned with other available MSHR entries; and 3) the blocked subsequent requests could have hit in the cache and thus do not need MSHRs.

Several works have been proposed to address reservation fails in some degree. Jia *et al.*[10] and Dai *et al.*[5] both use resource-aware cache bypassing techniques to bypass memory requests when they suffer stall in the cache. Xie *et al.*[34, 35] propose a compiler level cache bypassing technique. The compiler analyzes the cache utilization of a program based on the developed metric, and then selects certain instructions to access or bypass cache. While these cache bypassing techniques are effective in avoiding reservation fails when they are imminent, they do not explore the opportunity in improving miss handling architecture to reduce the likelihood of reservation fails in the first place. Another technique MRPB[10] is also proposed to actively reorder the requests into a cache-friendly order before accessing L1D cache and the associated MSHRs. Nevertheless, the effectiveness of MRPB is greatly limited by the "static" nature of MSHRs, e.g., when MSHR is entry-full (but not all the slots in the entries are occupied), no primary miss can be accepted even if these primary miss requests are perfectly reordered. Our proposed scheme addresses this issue by dynamically forming MSHR entries and slots from a pool of unified resources, thus complementing MRPB in a different way as shown in evaluation.

4 DYNAMICALLY LINKED MSHR

4.1 The Basic Idea

We propose *Dynamically Linked MSHR (DL-MSHR)*, a novel approach that allocates miss handling resources flexibly and adaptively. The basic idea is to decouple the static binding between a conventional MSHR entry and its constituting slots. Each DL-MSHR entry is dynamically formed from a pool of available slots. The adaptivity of DL-MSHRs is reflected in two aspects. Across applications, more entries with fewer slots are formed to meet the demand of primary-miss-predominant applications, whereas fewer entries but more slots per entry are formed for secondary-miss-predominant applications. Within an application, the number of slots that each entry has can also adapt to the frequency (demand) of memory accesses to the corresponding cache line. This flexibility allows DL-MSHRs to satisfy some extreme primary and secondary miss demands without the need for more physical entries or slots.

Figure 6 shows how DL-MSHRs integrate with other components of the system. At the high level, an array of DL-MSHRs replaces the conventional MSHR array to track multiple outstanding misses.

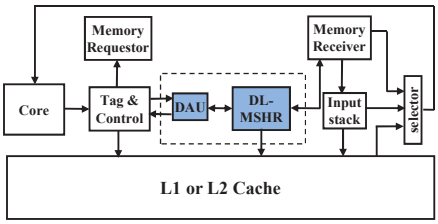


Figure 6: Overview of dynamically linked MSHRs (the new and modified components are highlighted).

A *Dynamic Allocation Unit (DAU)* is developed to control the operations of DL-MSHRs. The DAU is placed between the original Tag & Control unit and the DL-MSHR array. Upon a read or write request from the processing core or from the previous level in the memory hierarchy, the Tag & Control unit checks if the request hits in the cache. If not, the Tag & Control tries to insert the request to an MSHR and, if successfully (receiving acknowledgement from the MSHR), issues the request to the next memory level. With DL-MSHR, the DAU intercepts the signals from the Tag & Control and inserts the request to a dynamically linked DL-MSHR slot.

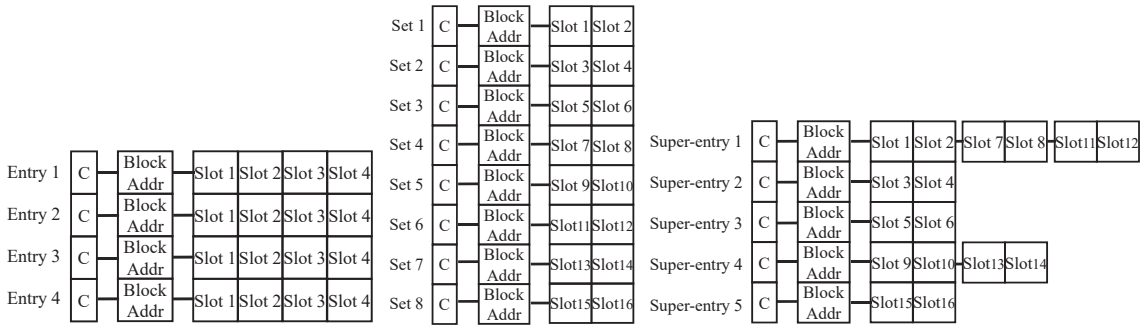
4.2 Challenges

While DL-MSHR conceptually is a simple but attractive approach, implementing miss handling entries that are flexible and adaptive faces several major challenges. First, unlike the linked list in data structure at the software level, where operations can be easily specified in high-level programming language and executed by a general-purpose processor, here dynamically linking slots needs to be implemented at the hardware level and controlled by a dedicated, low cost logic unit to handle various cases, which is not a straightforward task. Second, since MSHR slots are dynamically formed, additional time may be needed in finding available slots and in locating the position to link the slots. Thus, techniques and optimizations are needed to minimize the delay and power overhead of DL-MSHR, as well as to avoid frequent linking operations. Third, the DL-MSHR array and DAU should be designed in a way that is transparent to other components. In other words, all the original signals to and from the box with dashed boarder in Figure 6 should be exactly the same as in the conventional MSHR architecture. This avoids changes and verification efforts to other components, and helps to integrate the proposed scheme in commercial GPUs.

In the rest of this section, we present the detailed design of DL-MSHR, addressing what specific architecture changes are needed to link the slots, how the resources are organized physically and connected logically, what steps are involved in processing primary and secondary misses, when to allocate and free entries and slots, how the DAU is realized using finite state machines, along with four optimization techniques to further improve the efficiency of DL-MSHRs.

4.3 DL-MSHRs

Figure 6 illustrates the conventional explicitly addressed MSHRs and the proposed DL-MSHRs. All the slots in a row share the same block address and a comparator (denoted as "C"), and each slot includes the offset bits of the read/write data, a data buffer, and



(a) Conventional MSHRs (b) Physical view of DL-MSHRs (c) A possible logical state of DL-MSHRs

Figure 7: Illustration of conventional MSHRs and dynamically linked MSHRs (DL-MSHRs).

other related miss tracking information. Figure 7a shows a conventional MSHR architecture with 4 entries, each having 4 slots. As a result, if there are more than 4 concurrent primary misses or more than 3 concurrent secondary misses after any primary miss, there will be reservation fails due to MSHR entry-full and merge-full, respectively. However, it is unlikely that every cache line has the same number of outstanding secondary misses. Hence, many slots may still be available even in case of reservation fails.

To utilize the slot resource more efficiently, the proposed scheme decomposes the static entries into a pool of slots. Several slots form a slot *set* as the basic element for dynamic allocation (two slots in the example of Figure 7b). Managing resource at the granularity of sets rather than individual slots adds another level of flexibility and helps to reduce slot linking operations as discussed later. A slot set can be dynamically allocated as an independent entry for processing a primary miss, or can be linked after another slot set in an existing entry to increase the capacity of processing secondary misses, forming a "super-entry".

Figure 7b shows an example of how 8 slot sets are *physically* organized in the proposed DL-MSHR architecture, and Figure 7c shows one possible *logical* state at runtime. In this logical state, there are 5 super-entries or DL-MSHRs (we use the term super-entry and DL-MSHR interchangeably in this paper), and the super-entries have varying number of slots. Since there are 8 slot sets with 16 slots in total, maximally this DL-MSHR structure can process up to 8 outstanding primary misses concurrently if all the slot sets are assigned as entries, or handle up to 15 concurrent secondary misses after the primary miss if all the slot sets are linked together as one super-entry. This is significantly more than what conventional MSHRs in Figure 7a can handle.

The dynamic allocation is self-adaptive and does not require external interference to dictate when to link or delink. When a super-entry is full and a new secondary miss comes, it is the time to link a free slot set if one is available. When the requested data is returned from lower level and forwarded to the requesters, it is the time to break the corresponding super-entry and free all its slot sets. An internal control unit (i.e., DAU) is still needed to initiate the operations, and several extra bits are needed in each slot set:

Head bit (*H*): this bit indicates whether the slot set is the first set in a super-entry to handle a primary miss.

Linked bit (*L*): this bit indicates if there is another slot set attached to the current one to handle more secondary misses.

Pointer bits (*P*): these bits (e.g., 3 bits in the example of Figure 7c) work together with the *L* bit to specify the ID of the next linked set. This allows the control unit to find the physical location of the attached set.

Set free bit (*S_free*): this bit is set to 1 if all the slots in the current set are unoccupied, so the slot set can be dynamically allocated by the control unit.

Set full bit (*S_full*): this bit is set to 1 if all the slots in the current set of slots are occupied. If another secondary miss comes, a new slot set needs to be linked to the current set. The *S_free* and *S_full* bits are not mutually exclusive, e.g., when a slot set is partially occupied, both *S_free* and *S_full* bits are 0.

Lastly, a counter *nFreeSet* is maintained to track the number of free slot sets in the entire DL-MSHR structure. The counter is simply decremented or incremented whenever the control unit allocates or frees a slot set. Using the counter is a nice solution to avoid ANDing the *S_free* bit of every slot set, which would otherwise be slow. The above extra bits and the counter are all very small (no more than a few bits), which has minimal overhead compared with the slot set.

4.4 Operations

With the above architectural changes, we describe the three main operations of DL-MSHRs as follows.

Handling Primary Misses. When a miss is detected by the Tag & Control unit (TCU) in the conventional MSHRs, a search signal is sent to the comparator array to find whether there is a match in the MSHRs. The same signal is now sent to the DL-MSHRs. The block address included in the miss request is compared with the block address in each DL-MSHR. If no match is found (i.e., a primary miss), the *nFreeSet* counter is checked to see if any free slot set is available. If yes, an allocation signal is passed on to the DAU. A free slot set is selected to serve as a new super-entry that keeps track of the primary miss. The head bit is asserted to indicate the current set is an independent entry. The S_free bit is set to 0 signifying that this entry is currently occupied. The cache block address, offset address and requester ID are recorded in the first slot of the current set, as the slot set may consist of multiple slots. If this primary miss is a write request, the data is written into

the data buffer (applicable in the write-back cache). If the above *nFreeSet* counter is 0, it means that the entire DL-MSHR structure has no available slot set to handle any new primary miss. This memory request is stalled until a slot set becomes free, as indicated by *nFreeSet*.

Handling Secondary Misses. When the TCU detects a match in the conventional MSHRs, it generates a merge signal to the matching MSHR. This merge signal is now intercepted by the DAU. The DAU tries to merge the request into the matching DL-MSHR by storing the request in a free slot in the last slot set (i.e., tail slot set) of that super-entry. All the preceding slot sets should have been fully occupied. To locate the tail slot set, the DAU searches from the head slot set and follows the linked bit (*L*) and pointer bits (*P*) set by set until reaching the tail slot set, whose *L* bit should be 0. In the tail slot set, there are 3 possible cases:

(a) At least one slot is free (i.e., *S_full* is 0). In this case, the information of the miss request is recorded in the first available slot in the set. The DAU then modifies the *S_full* bit based on whether the current slot set is full after accepting this secondary miss. For a write miss, the DAU also writes the data into the data buffer.

(b) No slot is available in the tail set, but *nFreeSet* > 0. In this case, the DAU selects a free slot set to be linked as the new tail set. The old tail set stores the ID of the new tail set in the *P* bits and asserts the *L* bit to record the linking information. The new tail set deasserts its head bit, stores the miss request in the first slot (which should be free), and deasserts the the *S_free* bit.

(c) No slot is available in the tail set, and *nFreeSet* is also 0. In this case, no free slot set is available to be dynamically linked in the entire DL-MSHR structure. The miss request has to be stalled until a slot set becomes free, and then goes into the above case (b).

Deallocation of DL-MSHR. A super-entry and all of its slot sets are deallocated and recycled when the requested data is returned from lower memory levels and the data is forwarded back to the requesters. To deallocate, the DAU resets the *H*, *L*, *P*, *S_free* and *S_full* bits of all the slot sets in the super-entry. The *nFreeSet* counter is also incremented by the number of newly freed slot sets. Notice that, although the super-entry is deallocated, the data is still in the cache and subsequent accesses will result in cache hits.

4.5 Dynamic Allocation Unit (DAU)

A major task in implementing DL-MSHR is how to design a simple yet comprehensive control unit that can respond to various scenarios correctly and promptly. In this subsection, we present the design details of the *Dynamic Allocation Unit (DAU)*, which serves as an interfacing controller between the original Tag & Control unit and the DL-MSHR arrays. The key component in the DAU is a built-in finite state machine that controls various operations of slot sets. Figure 8 shows the finite state machine for a DL-MSHR example with two slots in a set[2]. Following state diagram conventions, the signals on the arrows are inputs and the signals inside the circles are outputs.

There are two main state-transfer paths in Figure 8. The lower path is activated by *ReqH* and the upper path is activated by *ReqA*.

[2]The minor states for error detection and fault control are omitted in this diagram for clarity, but they are all implemented. Additionally, the same state *S_0* is only replicated in the figure on both sides to avoid long arrows.

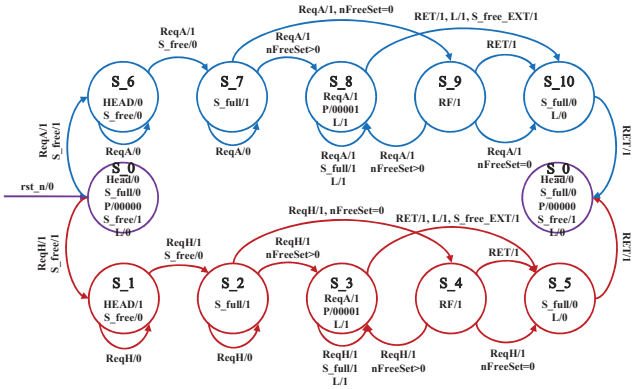


Figure 8: The finite state machine used to implement the Dynamic Allocation Unit (assuming 2 slots per set).

The *ReqH* is a signal that requests a free slot in a head set; if the set is not already a head set, the signal first marks the set as head and then requests a slot. Similarly, the *ReqA* is a signal that requests a free slot in an attached set; if the set is not already an attached set, the signal first transfers the state of the set to "attached".

The lower path containing *S_1* to *S_5* depicts the state-transfer when a free slot set becomes a head set. When *S_0* receives *ReqH* which results from a cache miss, this path is activated and the slot set becomes a head set. Hence, the Head bit is asserted, and the *S_free* bit is deasserted, as shown inside the circle of *S_1*. The *S_1* state implies that the first slot of the current set has recorded the information of a primary miss. At this point, if a new *ReqH* arrives requesting another free slot, the state is transferred to *S_2* and a secondary miss is recorded in the second slot of the current set. With a total of 2 free slots per set, the *S_full* signal should now be asserted. As more secondary misses continue to arrive at the current entry, the DAU checks the *nFreeSet* counter to see if there is any available slot set. If a free slot set is found, the state transfers to *S_3*, and the information of the linked set is recorded. During this state, a *ReqA* signal is sent out that marks the newly found set as "attached" (i.e., activating the upper path for that set, discussed shortly). However, if *nFreeSet* is 0, the state is directly transferred to *S_4* which generates a reservation fail (*RF*) signal. *S_4* may transfer back to *S_3* if a slot set becomes free, as indicated by *nFreeSet* > 0. Later, when the requested data is returned from the lower memory hierarchy, a *RET* signal is generated. This signal is used to release the occupied slots. This includes the transfer from both *S_3* and *S_4* to *S_5*. After the entire set is released, the state goes back to the initial state *S_0*.

Likewise, when *S_0* receives *ReqA* that requests the set to attach to another set, the upper path is activated. The *ReqA*, *RET* and *nFreeSet* are mainly responsible for driving the state transfer. To signify that the current set is used for holding secondary misses, the head bit is set to 0, and the *S_free* is also deasserted to denote an occupied slot, as shown in *S_6*. As more secondary misses arrive, the *S_full* is asserted, leading to *S_7*. Depending on whether a free slot set is available (i.e., if *nFreeSet* > 0), the state transfers to *S_8* or *S_9*. Later when the data is returned, the *RET* signal drives the state to *S_10* and the initial state.

Finally, for deallocation, when the state transfers to *S_0,* a *S_free* signal is asserted which serves as the external free signal *S_free_EXT* to the preceding slot set which is either another "attached" set (from *S_8* to *S_10*) or a "head" set (from *S_3* to *S_5*).

While the prior explanation of how DAU works is detailed, implementing the state diagram in Verilog HDL results in almost negligible hardware overhead of the control logic, as shown in evaluation (Section 6).

4.6 Additional Optimizations

Optimization 1: Group slots into sets. The above description of DL-MSHR started with grouped slots without too much explanation. In fact, this optimization has several benefits. First, grouping slots into one set can reduce hardware overhead, as most of the extra bits and resources in DL-MSHR are at the per set granularity. Second, grouping increases the chance of having a free slot when a secondary miss occurs, thus reducing the frequency of linking another slot set and the associated delay and power consumption. Third, grouping reduces the number of additional comparators needed by DL-MSHR. As each slot set can be used as a separate MSHR entry, the total number of comparators in DL-MSHR is equal to the number of slot sets. For example, in Figure 7b, with 2 slots per set, physically DL-MSHR needs 8 comparators. If there are 4 slots per set, the number of comparators would be the same as that of Figure 7a. Note that, even in this case, DL-MSHR is still better than Figure 7a because the slot sets can be dynamically linked.

Having more slots in a set increases the benefits of the above three aspects, but also reduces adaptivity. Our empirical study shows that having two slots per set offers a much better trade-off than other configurations by a large margin. Hence, two slots per set is used in this paper as the basic linking unit. In terms of the impact on the critical path, we have used Synopsys design compiler and CACTI 6.5 [19] to evaluate the CAM searching latency of different comparator configurations. A typical 32-entry MSHR design needs 0.2ns to complete the searching of 32 comparators (parallel searching but serial signal driving). When using 64 comparators such as in the 2-slot per set configuration, the searching time only increases slightly to 0.22ns, which is fast enough to match up with the frequency of most commercial GPUs.

Optimization 2: Disable unused comparators. Although physically each slot set has a comparator, the comparator is not used when the set is linked after another one. For example, logically only 5 comparators are active in Figure 7c. Therefore, the unused comparators can be disabled to avoid searching. To realize this, we can reuse the Head bit as a double-function bit. A deasserted Head bit in each slot set indicates that this set is either unused or attached to other set. In both cases, the associated comparator can be safely disabled by using the Head bit as a gated enabling signal. With this optimization, searching through the DL-MSHR arrays still takes roughly the same time (as all the Head bits still need to be searched), but the comparators of unused or attached sets are not activated, thus avoiding the associated power.

Optimization 3: Locate tail set faster. During the operation to link a slot set to an existing super-entry, the DAU needs to locate the tail slot set. If the super-entry contains many slot sets, this may take several cycles. To avoid this delay, an extra pointer that stores the ID of the current tail slot set can be added in the head set in a super-entry. The pointer is updated when a free slot set is linked as the new slot set, and is reset when the super-entry is deallocated. By adding this pointer, the latency to locate the tail set can be reduced to one cycle. We have evaluated this optimization, and simulation results show that the technique does improve performance, although the improvement is not large, around 0.5% IPC increase when averaged over the benchmarks. This is mainly because: 1) super-entries with a large number of linked slot sets are not common, 2) locating the tail set is needed only when linking slot sets, and 3) the latency can be partially hidden by multiple outstanding misses.

Optimization 4: Reserve head sets. We also augment the proposed DL-MSHR with the ability to reserve some head slot sets. In DL-MSHR, it is possible that all the slot sets are linked together as one huge super-entry to satisfy the need of a particular cache line with an unusual number of secondary misses. While this is intended and beneficial in some cases, it is rare that the entire many-core processor has only one primary miss. This can be easily addressed by setting the Head bit of some sets to always be 1 to prevent these sets from being attached to other slot sets. In our design, half of the slot sets are simply reserved to process primary misses, and the other half can be freely linked to other sets. Future work can be done along this interesting line to explore other configurations.

5 EVALUATION METHODOLOGY

We apply the proposed DL-MSHR to both L1D cache and L2 cache and implement in the cycle-accurate simulator GPGPU-Sim 3.2.2 [1]. Key parameters are listed in Table2. The L1D and L2 MSHR sizes are typical and in line with existing literature and products. Note that 32 entries are per SM for L1D and per bank for L2, so the GPU has thousands of MSHR entries as a whole. Different MSHR sizes (up to 256×32) are also evaluated to demonstrate the cost-effectiveness of DL-MSHR. The main evaluation assumes GTO warp scheduling policy, and other scheduling policies are also tested. GPUWattch [16] is employed to assess energy consumption.

A wide range of benchmarks from Rodinia [2], Parboil [29], and Nvidia GPU Computing SDK [24] are evaluated that include both compute and memory-intensive ones. Table 1 lists the details of the benchmarks. All the benchmarks are run to the end of their execution. It is important to note that memory coalescing in the SMs is already employed, so our evaluation methodology does not artificially increase the number of secondary misses to the cache. To evaluate hardware cost, we follow previous works (e.g. [10], [17], [31]) to use CACTI [19] to evaluate the "standard" parts of (DL-)MSHR. The data lines are stored in SRAM whereas the CAM structure is stored in flip-flops. All the new components such as the finite-state-machine in DAU and additional bits and control circuits are fully implemented in Verilog HDL and synthesized using Synopsys Design Compiler under NanGate FreePDK 45nm cell library [20] for more accurate area and power evaluation.

We compare the following 6 schemes: (1) **Baseline**: the baseline with conventional MSHRs shown in Table 2; (2) **2X_Entry**: doubling the number of MSHR entries of the Baseline (both L1D and L2); (3) **2X_Entry+2X_Slot**: doubling the number of entries and

Table 1: Evaluated benchmarks.

Benchmarks	Abbre.	Ref	Benchmarks	Abbre.	Ref
Backprop	BP	[2]	Transpose	TP	[24]
Bfs	BFS	[2]	Aligned Types	AT	[24]
B+tree	B+T	[2]	AsyncAPI	AA	[24]
Cfd	CFD	[2]	BlackSchole	BS	[24]
Dwt2d	DW	[2]	BinomialOptions	BO	[24]
Heartwall	HW	[2]	ConvolutionSeparable	CS	[24]
Hybridsort	HB	[2]	FastWalshTransform	FWT	[24]
Nw	NW	[2]	Merge Sort	MS	[24]
Srad	SRA	[2]	QR Generator	QG	[24]
StreamCluster	SC	[2]	Radix Sort Thrust	RST	[24]
Cutcp	CUT	[29]	Reduction	RED	[24]
Histo	HIS	[29]	ScalarProd	SP	[24]
Lbm	LBM	[29]	Scan	SCN	[24]
Stencil	STC	[29]	SobolQRNG	SQ	[24]
Sgemm	SG	[29]	Sorting Network	SN	[24]

Table 2: Simulator configuration.

# of SMs	28
Per-SM limit	48 warps, 8 CTAs
# of Mem partitions	8
L1D cache	16KB, 32-set, 4-way local write-back global write-through 32×8 MSHRs per SM (32 entries, 8 slots/entry)
L2 cache	8×128 KB, 64-set, 16-way allocate-on-miss, write-back 32×4 MSHRs per bank (32 entries, 4 slots/entry)
DRAM	FR-FCFS scheduler GDDR5, 16 banks peak bandwidth 345.6GB/s
SM/L2/DRAM clock	1137/1137/2700 MHz
Warp scheduler	GTO, LRR, Two-Level, SWL

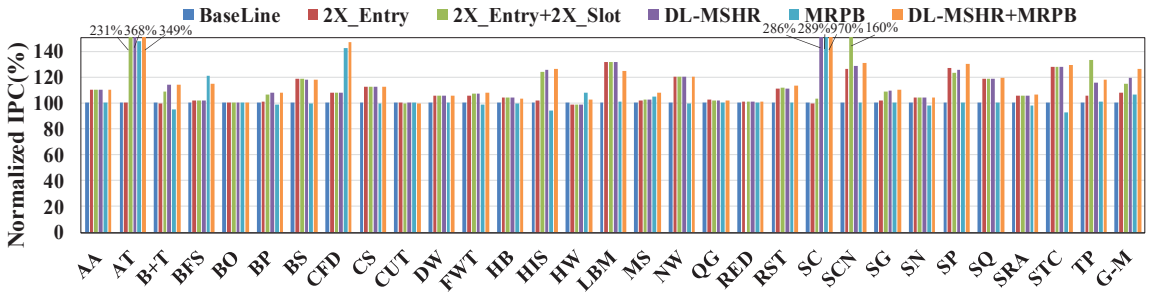


Figure 9: Performance comparison over the baseline architecture (normalized to the Baseline).

the number of slots of the baseline, i.e., 4X total slots as the Baseline; (4) **DL-MSHR**: the proposed DL-MSHR with the same total number of slots as the Baseline; (5) **MRPB**: a recent technique that reduces reservation fails by using Memory Request Prioritization Buffers to reorder memory requests in L1D cache and bypassing the cache for selected requests. Note that the prioritization signature used in MRPB is designed specifically for L1D and cannot be applied directly to L2. Also, the MRPB compared here includes both reorder and cache bypassing to improve its performance. (6) **DL-MSHR+MRPB**: applying DL-MSHR on top of MRPB to show that they exploit different opportunities and are complementary.

6 RESULTS AND ANALYSIS

6.1 Impact on Performance

Figure 9 compares the overall IPC improvement of different schemes over the baseline structure. Here the proposed DL-MSHR is applied to both L1D and L2 cache, and separate results are present in Section 6.3. Compared with the Baseline, when the number of entries is doubled, 2X_Entry improves the performance by 8.0% on average. When both entries and slots are doubled, 2X_Entry+2X_Slot improves the average performance by 14.5%. This shows that increasing the number of entries and/or slots can help to relieve some of the pressure on conventional MSHRs. However, some benchmarks such as *CS* and *DW* achieve IPC improvement only when entries are doubled, whereas some benchmarks such as *B+T* and *BP* gain performance only when slots are also doubled. These results are in line with our previous analysis that adding more entries or slots does not work well for all the benchmarks. In contrast, the number of entries and slots in DL-MSHR are dynamically determined based on the cache access patterns of different benchmarks. As a result, the proposed DL-MSHR scheme achieves the best performance among the first four schemes, with an average of 19.2% IPC improvement over the baseline architecture.

For *AT* and *SC*, they both benefit greatly from memory optimizations as *AT*'s kernel mostly consists of memory accesses and *SC*'s access pattern has very low reuse. However, 2X_Entry+2X_Slot does not improve much on SC because the burst secondary misses in SC demand dozens of slots with an entry, which the 2X Slots help marginally. In comparison, DL-MSHR is flexible and can attach up to 64 slots in an entry, thus meeting *SC*'s demand nicely. It is important to note that the above average IPC improvement is calculated based on geometric mean, so the performance improvement is not just because of a few very high bars. For example, among the 30 benchmarks in Figure 9, DL-MSHR has around 20% performance improvement for 8 benchmarks, with over 10% improvement for an additional 9 benchmarks. It is also worth mentioning that the average IPC of DL-MSHR is even 8.0% higher than that of 2X_Entry+2X_Slot which has 4X the number of total slots as DL-MSHR. This highlights the effectiveness and benefits of offering flexible resource allocation in DL-MSHR.

The MRPB compared in the evaluation also improves the geometric mean of IPC by 6.5%, showing that reordering memory requests and selectively bypassing cache help to reduce stalls when MSHRs are heavily used[3]. However, it does not help to balance the uneven slot utilization across different entries, and many resources are still idle even when entry-full or merge-full happens. This issue is addressed by employing the dynamically linked MSHRs. Therefore, the proposed DL-MSHR can be used to complement MRPB, and the

[3]The performance gain of MRPB here is different from the original paper as we also used Parboil and NVidia GPU Computing SDK benchmarks.

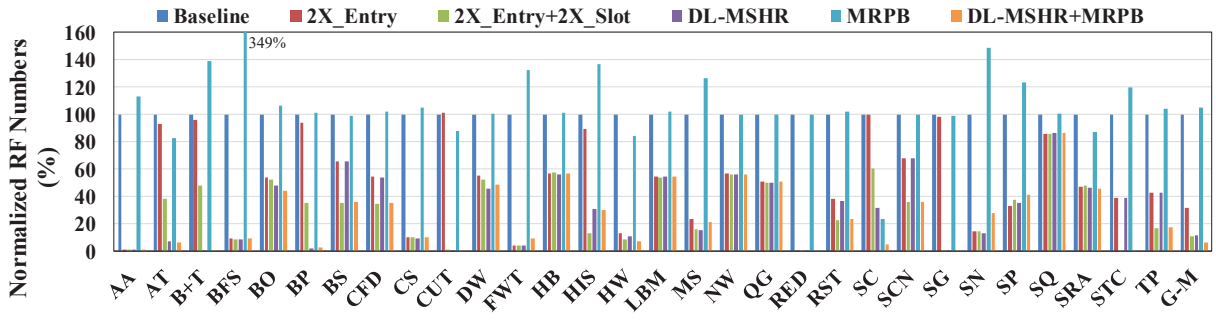


Figure 10: Reduction in the number of reservation fails.

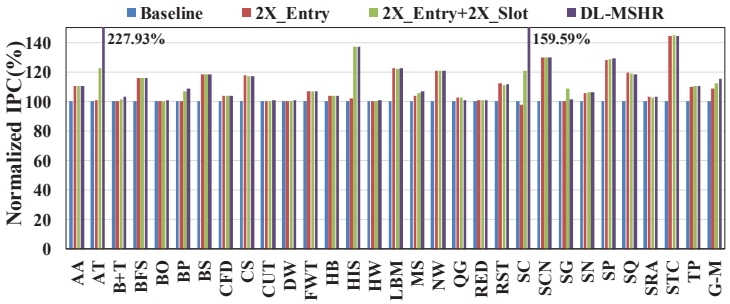


Figure 11: Performance comparison with L1D closed.

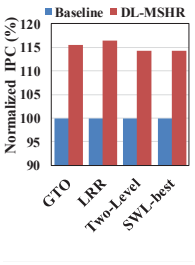


Figure 12: Schedulers.

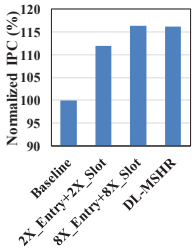


Figure 13: MSHR sizes.

resulting DL-MSHR+MRPB improves the IPC by 26.3%, on average, compared with the Baseline.

6.2 Reducing Reservation Fails

To provide more insights of the above performance impact, Figure 10 compares the number of reservation fails (RFs) normalized to the Baseline (the numbers also include RFs from other sources which accounts for less than 3% in the evaluated benchmarks). On average, doubling the number of entries (2X_Entry) reduces the RFs by 68.3%, and doubling the number of slots on top of this (2X_Entry+2X_Slot) decreases the RFs by 89.2%. In comparison, with the same number of slots as the Baseline, the proposed DL-MSHR can reduce the RFs by 88.1%, and is only slightly less than 2X_Entry+2X_Slot that has 4X the number of slots. It is interesting to see that DL-MSHR has a slightly smaller RF reduction but better IPC improvement than 2X_Entry+2X_Slot. The reason is that, the reduction in RFs is not proportional to increase overall performance, and varies among applications. For example, in the *LBM* benchmark, 2X_Entry+2X_Slot reduces 46.2% of the RFs and achieves 31.5% IPC improvement; whereas in *BP*, the same scheme reduces 64.2% of the RFs, but only increases IPC by 6.3%. These results indicate that the self-adaptive nature of DL-MSHR does not blindly optimize for the overall RF reduction, but rather fine-tunes the number of slots at the granularity of each entry to meet the need of primary and secondary misses at any specific time during execution.

Figure 10 also shows that MRPB does not directly reduce the number of RFs, which is expected as MRPB is not designed for that purpose. However, MRPB can help DL-MSHR to further bring down the number of RFs. This is shown in the last bar where RFs are reduced by 93.2%, on average, compared with the Baseline.

The large reduction of RF in DL-MSHR can be mainly attributed to the increase in MSRH utilization. Compared with the conventional MSHR, the proposed DL-MSHR improves MSHR utilization (calculated on a per slot basis) by 53.7% on average. The increase has been observed for every benchmark, although individual results are omitted here due to space limitation.

6.3 GPU Architecture Variation

In this subsection, we evaluate the impact of several GPU settings that may be different across GPU generations.

Closed L1 Data Cache. In some recent Maxwell and Pascal based GPUs, the L1D cache are closed (disabled) by default. To evaluate its impact, we disable L1D cache and apply DL-MSHR only to L2 cache. Figure 11 compares the performance. MRPB is no longer shown as it works on L1D. 2X_Entry, 2X_Entry+2X_Slot and DL-MSHR improve the Baseline IPC by 8.9%, 12.1% and 15.5% on average, respectively. Comparing Figure 9 and Figure 11, we can see that the benefits of DL-MSHR may come from both L1D cache and L2 cache, depending on memory access patterns:

(1) when memory requests from the L1D cache of different SMs converge into memory partitions (where L2 caches locate), the DL-MSHR in L2 brings majority of the benefits, e.g., for *AA* and *FWT*, disabling L1D cache only loses 0.2% IPC improvement;

(2) when there are many secondary misses in L1D but the requests for L2 does not exceed the capacity of MSHR in the L2 cache, the benefits mostly come from the DL-MSHR in L1D. For example, *B+T* gets 14.1% performance improvement when L1D is enabled, but the improvement drops to 3.4% when L1D is disabled;

(3) Some applications place pressure on both L1D and L2, and the DL-MSHR in both caches can help, e.g., for *AT*, DL-MSHR improves performance by 228.0% with L2 only, and achieves an additional 140.6% improvement when also applied to L1D.

Warp Scheduling Policies. In addition to closed L1D, warp scheduling policies may also vary a lot for different GPUs. Figure 12 compares the average IPC of benchmarks without DL-MSHR (first bar) and with DL-MSHR (second bar) for GTO, LRR, Two-level [21] and SWL-best [28]. We use SWL with the best static warp limiting numbers (SWL-best) to represent the oracle case for CCWS [28], OAWS [32]. As can be seen from the figure, different schedulers have some but limited impact on the effectiveness of DL-MSHR. In general, these and other schedulers can change the scheduled order and number of warps. This affects data locality and intensity to the cache which, in turn, change the hit and miss numbers. As our proposed scheme enhances miss handling, reduced cache misses may reduce the improvement of DL-MSHR. Nevertheless, cache misses are unavoidable even with the perfect warp scheduler, and warp scheduling does not help much in reducing reservation fails and increasing MSHR utilization. Thus, DL-MSHR consistently achieves sizable improvement under different warp schedulers, from 14.3% in SWL to 16.4% in LRR.

Different MSHR Sizes. While the MSHR sizes of L1D and L2 in our baseline are in line with prior work[10, 17], Figure 13 compares the effectiveness of DL-MSHR against other MSHR sizes. We assume 1-cycle access delay to all the conventional MSHR designs regardless of their sizes (thus representing the upper bound of VBF [18] or any other technique that reduces the access delay for large MSHRs); whereas DL-MSHR has 2-cycle access delay due to the access of super-entry and potentially linking of a new set (which is one cycle with the help of the tail set pointer, although this does not happen on every access). Hence, the comparison is slightly favored towards conventional MSHR designs. As shown in the figure, DL-MSHR with similar resource as Baseline is able to achieve nearly the same performance improvement as 8X_Entry+8X_Slot that has 64X resource of the Baseline, with an average IPC improvement of 16.2% vs. 16.4%. This indicates that the proposed DL-MSHR can be a cost-effective solution to realize very large MSHRs that may otherwise be needed in future GPUs.

Table 3: Area and Power of different MHA schemes.

Configuration	MHA Area(mm^2)	MHA Overhead	MHA Power(mW)	MHA Overhead
L1D: non-MHA area 0.11mm^2, non-MHA power 42.43mW				
Baseline	0.00386	3.51%	3.75	8.84%
2X_Entry	0.00739	6.72%	5.67	13.4%
2X_Entry+ 2X_Slot	0.0101	9.18%	7.20	17.0%
8X_Entry+ 8X_Slot	0.120	110%	57.3	135%
DL-MSHR	0.00449	4.08%	4.30	10.1%
MRPB	0.0126	11.5%	15.2	35.9%
L2: non-MHA area 0.97mm^2, non-MHA power 343.28mW				
Baseline	0.0601	6.19%	35.3	10.3%
2X_Entry	0.114	11.7%	63.9	18.6%
2X_Entry+ 2X_Slot	0.216	22.3%	114	33.4%
8X_Entry+ 8X_Slot	3.18	328%	1500	437%
DL-MSHR	0.0611	6.30%	36.1	10.5%

6.4 Area and Power Overhead

Table III summarizes the area and power overhead of different schemes. The results are obtained from Cacti 6.5 and Synopsys Design Compiler. The additional overhead of DL-MSHR over conventional MSHR comes from the extra status bits (head bit, linked bit, pointer bits, set free and full bits), additional comparators and block address fields, a free slot set counter, and the DAU control unit. The overhead of MRPB is mainly from the reorder buffers and related control logics.

To understand the relative impact of hardware cost on the cache subsystem, we put the area and power of the non-MHA part (i.e., the regular tag and data part) of L1D and L2 on top of each table section, whereas the numbers in the main table refer to the MHA part (i.e., MSHRs, comparators, controls, etc.). For instance, the MHA of Baseline in L2 incurs 0.0601mm^2, which is equivalent to 6.19% of the non-MHA part of L2 area. As can be seen, the area and power overhead of directly increasing MSHR sizes quickly becomes substantial, accounting for a significant percentage of regular cache (e.g., 2X_Entry+2X_Slot has 22.3% of L2 area). In comparison, the area and power of the proposed DL-MSHR is very close to the MHA part of Baseline, e.g., within around 0.56% area of Baseline for L1D and within around 0.11% area of Baseline for L2. When taking the previous performance results into consideration, it can be seen that, compared with other optimization schemes, DL-MSHR has higher performance *and* lower area and power overhead.

6.5 Impact on Energy

Due to the small hardware overhead, DL-MSHR has minimal impact on the power consumption of GPUs. Therefore, the energy consumption is mainly reduced because of the shorter execution time for reduced static energy. GPUWattch results show that, compared with the Baseline, the proposed DL-MSHR achieves an overall GPU energy savings of 15.7% on average. In comparison, 2X_Entry, 2X_Entry+2X_Slot, and MRPB reduce the energy consumption by 8.7%, 1.43% and 5.1%, respectively. It is also interesting to see that, compared with 2X_Entry, the 2X_Entry+2X_Slot consumes more total energy even though it has shorter execution time. This is because providing additional MSHR resources in 2X_Entry+2X_Slot taxes on the static energy, which illustrates from the energy perspective that naively adding MSHR resources is not an ideal option.

7 CONCLUSION

Contemporary GPUs have an increasing demand for higher memory level parallelism. Consequently, the miss handling architecture must be designed to efficiently track a large number of outstanding memory requests concurrently. In this paper, we propose a dynamically linked MSHR (DL-MSHR) architecture, which forms MSHR entries dynamically to adapt to application primary and secondary miss behaviors. Evaluation shows significant reduction in reservation fails and large improvement in overall performance, while incurring much less area and power overhead than the alternatives. These results demonstrate the viability and potential benefits of dynamic MSHR structures.

8 ACKNOWLEDGMENTS

We sincerely thank the reviewers for their helpful comments and suggestions. This research was supported, in part, by the National Science Foundation grants #1566637, #1619456, #1619472 and #1750047.

REFERENCES

[1] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*.

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*.

[3] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*.

[4] Intel Core. 2010. i7 Processor Series Datasheet.

[5] Hongwen Dai, Christos Kartsaklis, Chao Li, Tomislav Janjusic, and Huiyang Zhou. 2014. RACB: Resource Aware Cache Bypass on GPUs. In *Intl. Symp. on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*.

[6] KI Farkas and NP Jouppi. 1994. Complexity/performance tradeoffs with non-blocking loads. In *Intl. Symp. on Computer Architecture (ISCA)*.

[7] Yongbin Gu and Lizhong Chen. 2018. CART: Cache Access Reordering Tree for Efficient Cache and Memory Accesses in GPUs. In *Intl. Conf. on Computer Design (ICCD)*.

[8] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Intl. Symp. on Computer Architecture (ISCA)*.

[9] Magnus Jahre and Lasse Natvig. 2011. A high performance adaptive miss handling architecture for chip multiprocessors. In *Transactions on High-Performance Embedded Architectures and Compilers IV*.

[10] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*.

[11] Naifeng Jing, Yao Shen, Yao Lu, Shrikanth Ganapathy, Zhigang Mao, Minyi Guo, Ramon Canal, and Xiaoyao Liang. 2013. An energy-efficient and scalable eDRAM-based register file architecture for GPGPU. In *Intl. Symp. on Computer Architecture (ISCA)*.

[12] Mahmoud Khairy, Mohamed Zahran, and Amr G Wassal. 2015. Efficient utilization of GPGPU cache hierarchy. In *Workshop on General Purpose Processing using GPUS (GPGPU)*.

[13] Youngsok Kim, Jaewon Lee, Jae-Eon Jo, and Jangwoo Kim. 2014. GPUdmm: A high-performance and memory-oblivious GPU architecture using dynamic memory management. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*.

[14] David Kroft. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Intl. Symp. on Computer Architecture (ISCA)*.

[15] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. 2015. CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. In *Intl. Symp. on Computer Architecture (ISCA)*.

[16] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. In *Intl. Symp. on Computer Architecture (ISCA)*.

[17] Lingda Li, Ari B Hayes, Shuaiwen Leon Song, and Eddy Z Zhang. 2016. Tag-split cache for efficient GPGPU cache utilization. In *Intl. Conf. on Supercomputing (ICS)*.

[18] Gabriel H Loh. 2008. 3D-stacked memory architectures for multi-core processors. In *Intl. Symp. on Computer Architecture (ISCA)*.

[19] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* (2009).

[20] Inc NanGate. 2017. Nangate freePD45 open cell library. *Avalible at: http://http://www.nangate.com/* (2017).

[21] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Intl. Symp. on Microarchitecture (MICRO)*.

[22] NVIDIA Nsight and Visual Studio Edition. 2013. 3.0 User Guide. *NVIDIA Corporation* (2013).

[23] Nvidia. 2014. NVIDIA GeForce GTX 980 White Paper.

[24] GPU Nvidia. 2013. Computing SDK. *Gpu computing sdk, Avalible at: https://developer. nvidia. com/gpu-computing-sdk* (2013).

[25] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[26] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Intl. Symp. on Microarchitecture (MICRO)*.

[27] Moinuddin K Qureshi, David Thompson, and Yale N Patt. 2005. The V-Way cache: demand-based associativity via global replacement. In *Intl. Symp. on Computer Architecture (ISCA)*.

[28] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *Intl. Symp. on Microarchitecture (MICRO)*.

[29] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* (2012).

[30] Yingying Tian, Sooraj Puthoor, Joseph L Greathouse, Bradford M Beckmann, and Daniel A Jiménez. 2015. Adaptive GPU cache bypassing. In *Workshop on General Purpose Processing using GPUS (GPGPU)*.

[31] James Tuck, Luis Ceze, and Josep Torrellas. 2006. Scalable cache miss handling for high memory-level parallelism. In *Intl. Symp. on Microarchitecture (MICRO)*.

[32] Bin Wang, Yue Zhu, and Weikuan Yu. 2016. OAWS: Memory Occlusion Aware Warp Scheduling. In *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*.

[33] Intel Xeon. 2012. E5 Processor Series Datasheet.

[34] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An efficient compiler framework for cache bypassing on GPUs. In *IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD)*.

[35] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*.