EXTENDING CACHE REPLACEMENT POLICIES OF GPGPU-Sim (SRRIP, BRRIP, LFU)

Saurav Kumar, Bharat Kalyanaraman, Snehal Patil
Department of Electrical and Computer Engineering
North Carolina State University, Raleigh, NC, USA

***Abstract*— With the requirement for higher performance and barriers such as ILP wall, Memory wall and Power wall, heterogeneous systems have found their way into wide variety of applications in both user and commercial domain. They offer more computing resources while keeping the power considerations in manageable limits. Multiple challenges have cropped up due to the shift to concurrent programming and widespread adoption of these systems. These include shared resource contention, performance challenges, energy efficiency, fairness and throughput. Thread level parallelism (TLP) aware policies try to mitigate effect of cache contention in a heterogeneous architecture. This paper describes the motivation, design and results of our implementation of replacement policies in GPGPU-Sim in order to create a baseline for simulating performance in heterogeneous architecture. We implement SRRIP, BRRIP and LFU policies in GPGPU-Sim and noticed that the LFU policy performs similar to LRU since the aging algorithm implemented is quite aggressive. SRRIP and BRRIP perform better for cache-friendly workload but lose out in memory intensive benchmarks. The LFU policy without aging was not able to match LRU in most cases in terms of miss rate; hence aging was introduced in LFU replacement policy.**

I. INTRODUCTION

Caches are critical shared resources in any processor architecture as they subside the long latencies of memory accesses. Classic cache management policies work fine with single core architecture but attention is needed while implementing them on GPGPU type architecture. Different policies were proposed keeping in mind different factors affecting application throughput. This project aims at implementing one such policy [5] called Re-reference Interval Prediction Policy cache management. LFU is also implemented and studied to compare the relative performances on different benchmarks.

Shared resources in heterogeneous architecture include shared LLC, DRAM controller, front side bus and prefetching hardware. However, in recent processor designs, the prefetchers have been made private and are located at L2 cache and the DRAM controllers are moved on die and are more sophisticated thereby reducing the sharing effects [2]. LLC management thus, is of key importance as it significantly affects performance of the application as well as overall system throughput. GPGPU applications tend to have much higher memory access rates in the order of 300 requests per kilocycles than the existing CPU applications and although the memory latencies are handled by thread level parallelism in GPGPUs, their higher access rate may mar the performance of concurrent CPU applications due to cache pollution [1]. Existing cache management policies when implemented on GPGPUs result in non-temporal data storage leading to degraded cache performance. Promotion based cache management schemes are adopted to alleviate this problem.

RRIP, which is a type of promotion, based cache management, approximates the accesses as near, long or distant future and tries not to replace lines, which are predicted to be reused in the near future. It is made highly resistant to non-temporal access patterns and thrashing applications by assigning shorter lifetime to each cache block and then promoting them to higher order only upon hits. There are two flavors of this proposed RRIP [5], first is SRRIP i.e. static RRIP and the other is BRRIP i.e. bimodal RRIP. In SRRIP, each new incoming block is inserted at a position with RRPV (Re-reference prediction value) of 2^n-2, where n is the cache associativity, while for BRRIP it is inserted at 2^n-2^{th} position with a probability of 5% and for the remaining cases the incoming block is inserted at 2^n-1 position. RRIP then dynamically selects any of these two policies based on the application's performance. LFU on the other hand, maintains a counter per block which gets incremented upon hit to check access frequency.

II. RELATED WORK

RRIP [5] i.e. RE-Reference Prediction Interval Policy and UCP [6] i.e. Utility based Cache Partitioning were introduced to overcome the problems faced by existing LRU policy. RRIP works out the solution for bad cache performance under thrashing applications or applications using large working set while UCP enables high cache performance for concurrent applications with very low overhead. While both these policies are meant to improve shared cache performance, they fail to serve the very purpose when implemented on heterogeneous architecture like on chip CPU-GPGPUs. Such a hetero-architecture needs to address two main problems: 1] Cache unfriendliness of some GPGPU applications 2] Cache Interference introduced by GPGPU applications. These two problems can be solved to some extent by TAP-UCP and TAP-RRIP, proposed by Lee & Kim [1], which are TLP (thread level parallelism) aware versions of UCP and RRIP mentioned before.

These TAP versions use Core Sampling (to verify cache-friendliness of GPGPU applications) and Cache Block Lifetime Normalization (to take access frequency gap between CPU and GPGPU applications into account) techniques to modulate the cache usage of GPGPU applications which run concurrently with on chip CPU applications, based on the performance data which is generated periodically. While TAP-UCP partitions the cache space among different applications after arbitrating through various possible optimal solutions, TAP-RRIP dynamically adapts between SRRIP and BRRIP schemes to get optimal performance. In order to aid analysis of the TAP policies using GPGPU-sim, the baseline policies have to be introduced in the source code. We made the necessary changes to provide three selectable cache replacement policies namely, LFU, SRRIP and BRRIP.

III. SIMULATOR INFRASTRUCTURE

The specifics of how GPGPUSim v3.2.0 implements the cache hierarchy in GPU's, is described in this section. gpu-cache.h implements all the caches used by the ldst_unit (load store unit). Fig 1 shows the hierarchy.

1) cache_t
A high level cache that just declares the virtual "access" function.
2) baseline_cache:
This class implements the basic cache functionality. It contains functions like:
- fill()
- waiting_for_fill()
- access_ready()
- flush()
- print(), accesses and misses
- display_state()
- get_data_stats()
- Note: Each of the subclasses implement their own "access" function. Hence it is a virtual function in this class

3) read_only_cache:
This class inherits from baseline_cache. It's just a container class.This just has a virtual function defined for "access".
4) data_cache:
Again this class inherits from the class baseline_cache. It implements the common functions for L1 and L2 caches. The various functionalities offered by this class are as follows:
- Functions for read/write hit/misses.
- Depending on the configuration, WriteHit Writeback, WriteHit WriteThrough, WriteMiss Allocate, WriteMiss NoAllocate.

5) l1_cache
This class is inherited from data_cache. It implements the "access" functionality. This is the same function that was declared as virtual in the class baseline_cache.
6) l2_cache
Again this class is inherited from the class data_cache. It implements the "access" functionaluity that was declared as Virtual in the class baseline_cache.
7) tex_cache
Implements the texture cache. Inherits directly from cache-t. Models the functionality associated with Texture caches.

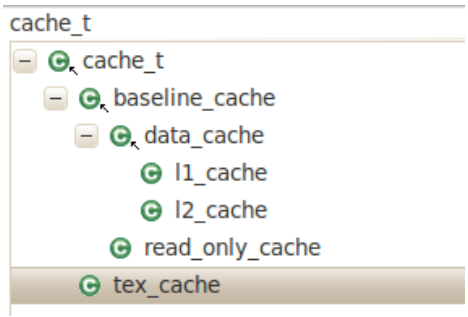

Figure 1: Cache Hierarchy in GPGPU-Sim

Additional details:
The gpu-cache.h file implements various struct's, enumerations for modelling the cache. The enums's are as follows. The naming signifies the use of the enum.

1. cache_block_state
2. cache_request_state
3. cache_event
4. replacement_policy_t
5. write_policy_t
6. allocation_policy
7. write_allocate_policy
8. mshr_config_t

The actual values that are enumerated are clearly name and can be found in the gpu-cache.h file.

The supporting structs and classes that are used are as follows.

- struct cache_block_t
  Contains all the specifics related to a cache block.
- class cache_config
  Has the specifics regarding the cache configuration, replacement policy, write policy, allocation policy, etc.
- class tag_array
  An array of cache_block_t's
- class mshr_table
  Miss status handling register

There are two major functions in the implementation of caches in GPGPUSim.
1. probe()
checks for a block address without affecting the LRU position counters/timestamps.
2. access()
models a lookup that affects the LRU positioning counter.

Miss Status handing register (MSHR):
MSHR's are modeled with the mshr_table class emulates a fully associative table with a finite number of merged requests. Requests are released from the MSHR through thenext_access() function.

Other details:
The read_only_cache class is used for the constant cache and as the base-class for thedata_cache class. This hierarchy can be somewhat confusing because R/W data cache extends from theread_only_cache. The only reason for this is that they share much of the same functionality with the exception of the access function which deals has to deal with writes in the data_cache. The L2 cache is also implemented with the data_cache class.

The tex_cache class implements the texture cache outlined in the architectural description above. It does not use the tag_array or mshr_table since it's operation is significantly different from that of a conventional cache.

IV. CODE IMPLEMENTATION

Changes were done in the files "gpu-cache.cc" and "gpu-cache.h", in order to implement LFU, SRRIP and BRRIP policies. Other source files in gpgpu-sim src directory instantiate caches and use the policies, which are implemented, in the files we modified. Hence, a thorough understanding of the files/caches which relied on these policies and configuration options was necessary in order to extend current

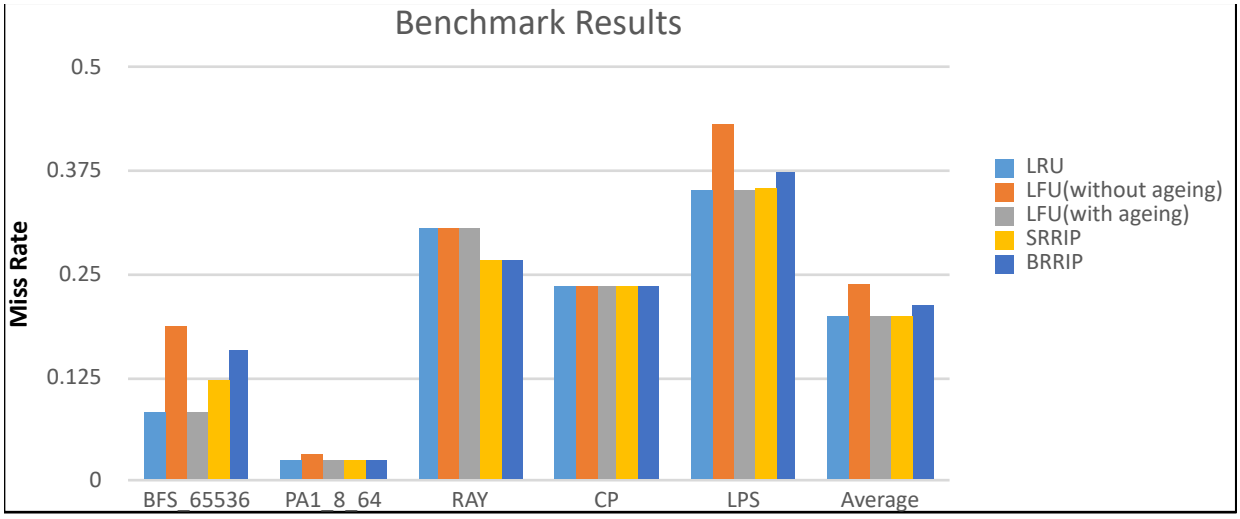

Figure 2: Benchmark Results

policies. Entries for new replacement policies were registered in the replacement_policies_t enumerated data type.

The write_policy_t enumerated data type uses switch case statements to decipher selected eviction policies after parsing the "gpgpusim.config" file. Hence, options to select the new policies were introduced here. The policy selection is now done using the following keywords:

L LRU
F FIFO
S SRRIP
B BRRIP
H LFU

A. SRRIP

A variable m_rrvp was introduced in the cache_block_state to aid the decision making for RRIP policies. This variable is initialized to maximum possible integer value. The change done to select an eviction candidate based on newly introduced policies is the probe function. Under SRRIP and BRRIP, the cache line with maximum RRVP is selected for eviction. The RRVP of all other lines is updated to a new RRPV value based on the difference between maximum RRPV value and (2^{assoc} – 1). A cache line that hits gets it's RRPV value updated in the access function.

B. BRRIP

Since BRRIP inserts a new cache line in the 2^{assoc} - 2 position with a low probability (usually 5%) and at 2^{assoc} -1 position otherwise, a global macro for the probability selection was included in the header file. In order to generate boolean values with the required probability a function nextBool [4] was used which employs the rand function and RAND_MAX variable defined in the stdlib header. Replacement and updating values of RRPV variable is done similar to SRRIP.

C. LFU

Recording the hit rate for each line requires introducing a variable m_hits in the cache_block_state enumerated data type. Previously implemented policies (LRU and FIFO) used the last access time or allocation time in order to identify block eligible for eviction. Hence, no existing counters could be reused. The m_hits variable is initialized to zero. LFU policy selects the block with least number of hits for eviction. If multiple blocks have same number of hits then the LRU policy or the timestamp is used to determine the candidate for eviction. The hit count for a block is updated in the access function. Additionally, in the access function, total hits for each line in a particular set is reduced by half at every access. This is done in order to implement ageing and avoid highly referenced stale blocks becoming resident in sets, which might degrade performance.

V. Results and Analysis

We ran five different workloads in GPGPU-sim with the newly introduced policies. Configurations corresponding to NVIDIA GeForce GTX 480 were used for the benchmark runs. IPC is used to determine performance improvement in heterogeneous systems. However, since the replacement policies are being introduced and tested only for GPGPU, L2 cache miss rate was used to determine efficiency of new policies.

Table 1 shows the baseline configuration used in our experiments. We model the proposed replacement policy on GPGPU-Sim v3.2.0 with GPUWattch enabled.

Type	Configuration
GPU	nVidia GTX480
L1 Cache	48kb, 32 sets, 128 kb block size, LRU
L2 Cache	786kb, 64 sets, 128 kb block size

The results that were obtained are shown in Figure 2.

From the results, we can observe that for cache sensitive applications/loads (e.g. RAY), RRIP works better than other policies. Whereas the other benchmarks are not cache sensitive and we observe that there is not much benefit over LRU for the different policies. And one more important point to note is that LFU policy implemented without aging consistently performs worse than all other benchmarks. Also, we believe that the LFU with aging has been implemented with a very aggressive aging policy. Thus its performance is very comparable to LRU. If the aggressiveness in aging algorithm were to be reduced, we believe that it could perform better than LRU. Overall we believe that CPU last level cache management policies can be translated to GPU's and that GPU's will benefit from these policies in case of workloads that are cache sensitive.

VI. Future Work

The replacement policies were modeled and tested for benchmarks from various categories. However, further detailed benchmarking on other varied benchmarks can be performed to reinforce the results. The essence of our approach is that the Last level GPU cache (here the L2 cache) can be viewed as a shared resource between the various GPU cores (SM's). Hence logically, the replacement policies that are applied to shared CPU LLC's should provide better results than the simple LRU policy. For this purpose, we implemented LFU, SRRIP and BRRIP. The next logical step would be to implement a Policy selector mechanism that would monitor the performance of the GPU under all of these different policies and then select one policy, which would work best based on monitored values. This would involve having monitors on each of the GPU cores, running different cores on different policies and then selecting the best performing policy.

VII. Conclusion

With multi-cores and heterogeneous architectures becoming ubiquitous in computing, concurrent execution of many applications has become quite popular. This causes contention for shared resources. Since the on-chip LLC is at the periphery of resources on the multi-core processor, it is one of the last avenues to tackle majority of delays before the application issues a request to the slower main-memory. Researches have proposed policies such as UCP and RRIP for LLC management in multi-core systems. Another work that derives from these policies and targets heterogeneous architectures uses TLP aware mechanisms over UCP and RRIP. GPGPU-Sim, a cycle level GPU performance simulation currently provides FIFO and LRU cache replacement policies. In order to allow analysis of the newly proposed techniques in heterogeneous systems, replacement policies in GPGPU-Sim need to be extended. We introduced the policies SRRIP, BRRIP and LFU in current source code of GPGPU-sim and evaluate their performance using BFS, RAY, CP, LPS and the PA1 implementation. We conclude that LLC management policies can be used in GPU's with some modifications and it will result in performance improvements for cache sensitive workloads.

References

1. Jaekyu Lee; Hyesoon Kim, "TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture," High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on, vol., no., pp.1, 12, 25-29 Feb. 2012.
2. Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. 2012. CRUISE: cache replacement and utility-aware scheduling. In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII). ACM, New York, NY, USA, 249-260.
3. Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, April 19-21, 2009.
4. How to generate a boolean with p probability using c rand function. http://stackoverflow.com/questions/3771551/how-to-generate-a-boolean-with-p-probability-using-c-rand-function
5. A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In ISCA-32, pages 60–71, 2010.
6. M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In MICRO-39, pages 423– 432, 2006.