In the proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA), February 2012

TAP: A TLP-Aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture

Jaekyu Lee Hyesoon Kim
School of Computer Science
Georgia Institute of Technology
{jaekyu.lee, hyesoon}@cc.gatech.edu

Abstract

Combining CPUs and GPUs on the same chip has become a popular architectural trend. However, these heterogeneous architectures put more pressure on shared resource management. In particular, managing the last-level cache (LLC) is very critical to performance. Lately, many researchers have proposed several shared cache management mechanisms, including dynamic cache partitioning and promotion-based cache management, but no cache management work has been done on CPU-GPU heterogeneous architectures.

Sharing the LLC between CPUs and GPUs brings new challenges due to the different characteristics of CPU and GPGPU applications. Unlike most memory-intensive CPU benchmarks that hide memory latency with caching, many GPGPU applications hide memory latency by combining thread-level parallelism (TLP) and caching.

In this paper, we propose a TLP-aware cache management policy for CPU-GPU heterogeneous architectures. We introduce a core-sampling mechanism to detect how caching affects the performance of a GPGPU application. Inspired by previous cache management schemes, Utility-based Cache Partitioning (UCP) and Re-Reference Interval Prediction (RRIP), we propose two new mechanisms: TAP-UCP and TAP-RRIP. TAP-UCP improves performance by 5% over UCP and 11% over LRU on 152 heterogeneous workloads, and TAP-RRIP improves performance by 9% over RRIP and 12% over LRU.

1. Introduction

On-chip heterogeneous architectures have become a new trend. In particular, combining CPUs and GPUs (for graphics as well as data-parallel applications such as GPGPU applications) is one of the major trends, as can be seen from Intel's recent Sandy Bridge [7], AMD's Fusion [2], and NVIDIA's Denver [15]. In these architectures, various resources are shared between CPUs and GPUs, such as the last-level cache, on-chip interconnection, memory controllers, and off-chip DRAM memory.

The last-level cache (LLC) is one of the most important shared resources in chip multi-processors. Managing the LLC significantly affects the performance of each application as well as the overall system throughput. Under the recency-friendly LRU approximations, widely used in modern caches, applications that have high cache demand acquire more cache space. The easiest example of such an application is a streaming application. Even though a streaming application does not require caching due to the lack of data reuse, data from such an application will occupy the entire cache space under LRU when it is running with a non-streaming application. Thus, the performance of a non-streaming application running with a streaming application will be significantly degraded.

To improve the overall performance by intelligently managing caches, researchers have proposed a variety of LLC management mechanisms [4, 8–10, 21, 22, 27, 28]. These mechanisms try to solve the problem of LRU by either (1) logically partitioning cache ways and dedicating fixed space to each application [10,21,22,28] or (2) filtering out adverse patterns within an application [9, 27]. In logical partitioning mechanisms, the goal is to find the optimal partition that maximizes the system throughput [21, 22, 28] or that provides fairness between applications [10]. On the other hand, the other group of cache mechanisms identifies the dominant pattern within an application and avoids caching for non-temporal data. This can be done by inserting incoming cache blocks into positions other than the most recently used (MRU) position to enforce a shorter lifetime in the cache.

However, these mechanisms are not likely applicable to CPU-GPU heterogeneous architectures for two reasons. The first reason is that GPGPU applications often tolerate memory latency with massive multi-threading. By having a huge number of threads and continuing to switch to the next available threads, GPGPU applications can hide some of the off-chip access latency. Even though recent GPUs have em-

ployed hardware-managed caches [14], caching is merely a secondary remedy. This means that caching becomes effective when the benefit of multi-threading is limited, and increasing the cache hit rate even in memory-intensive applications does not always improve performance in GPGPU applications. The second reason is that CPU and GPGPU applications often have different degrees of cache access frequency. Due to the massive number of threads, it is quite common for GPGPU applications to access caches much more frequently than CPUs do. Since previous cache mechanisms did not usually consider this effect, many policies will favor applications with more frequent accesses or more cache hits, regardless of performance.

To accommodate the unique characteristics of GPGPU applications running on heterogeneous architectures, we need to consider (1) how to identify the relationship between cache behavior and performance for GPGPU applications even with their latency-hiding capability and (2) the difference in cache access rate. Thus, we propose a thread-level parallelism (TLP)-aware cache management policy (TAP). First, we propose *core sampling* that samples GPU cores with different policies. For example, one GPU core uses the MRU insertion policy in the LLC and another GPU core uses the LRU insertion. Performance metrics such as cycles per instruction (CPI) from the cores are periodically compared by the *core sampling controller* (CSC) to identify the cache friendliness[1] of an application. If different cache policies affect the performance of the GPGPU application significantly, the performance variance between the sampled cores will be significant as well. The second component of TAP is *cache block lifetime normalization* that considers the different degrees in access rate among applications. It enforces a similar cache lifetime to both CPU and GPGPU applications to prevent adverse effects from a GPGPU application that generates excessive accesses.

Inspired by previously proposed Utility-based Cache Partitioning (UCP) and Re-Reference Interval Prediction (RRIP) mechanisms, we propose two new mechanisms, TAP-UCP and TAP-RRIP, that consider GPGPU application characteristics in heterogeneous workloads.

The **contributions** of our paper are as follows:

1. To the best of our knowledge, we are the first to address the cache-sharing problem in CPU and GPU heterogeneous architectures.
2. We propose a core sampling mechanism that exploits the symmetric behavior of GPGPU applications.
3. We propose two new TLP-aware cache management mechanisms, TAP-UCP and TAP-RRIP, that significantly improve performance across 152 heterogeneous workloads.

[1] *Cache friendliness* means that more caching improves the performance of an application.

2. Background and Motivation

2.1. Target Architecture

As Figure 1 illustrates, we project that future heterogeneous architectures will have high-end CPUs and GPUs on the same chip, with all cores sharing various on-chip resources. Intel's Sandy Bridge [7] is the first commercial processor that integrates GPUs on the chip, and AMD's Fusion architecture [2] integrates more powerful GPUs on the same die. In these architectures, we can run OpenCL [16] like applications that utilize both types of cores or multi-program workloads that run different applications on CPUs and GPUs. In this paper, we focus on multi-program workloads on heterogeneous architectures.

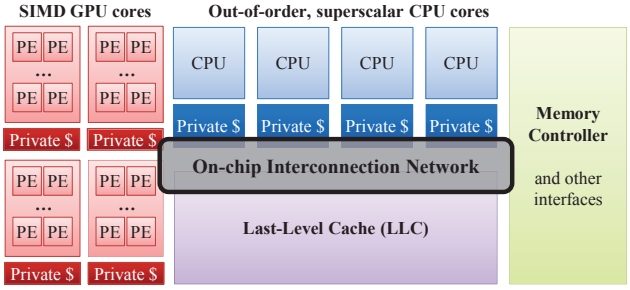


Figure 1. Target heterogeneous architecture.

Since these two different types of cores share on-chip resources, the resource-sharing becomes a new problem due to the different nature of the two types of cores. Thus, we aim to solve the problem in these new architectures.

2.2. Characteristic of GPGPU Applications

In this section, we explain the memory characteristics of GPGPU applications. First, we classify GPGPU applications based on how the cache affects their performance. Figures 2 and 3 show cycles per instruction (CPI) and misses per kilo instruction (MPKI) variations for all application types as the size of the cache increases. Note that to increase the size of the cache, we fix the number of cache sets (4096 sets) and adjust the number of cache ways from one (256 KB) to 32 (8 MB). Application types A, B, and C in Figure 2 can be observed in both CPU and GPGPU applications. We summarize these types as follows:

- Type A has many computations and very few memory instructions. The performance impact of those few memory instructions is negligible since memory latencies can be overlapped by computations. Thus, the CPI of this type is close to the ideal CPI, and MPKI is also very low.
- Thrashing applications are typical examples of type B. Because there is a lack of data reuse or the working set size is significantly larger than the limited cache size, CPI is high and MPKI is extremely high.
- Type C applications are typical cache-friendly benchmarks. For these benchmarks, more caching improves the cache hit rate as well as performance.

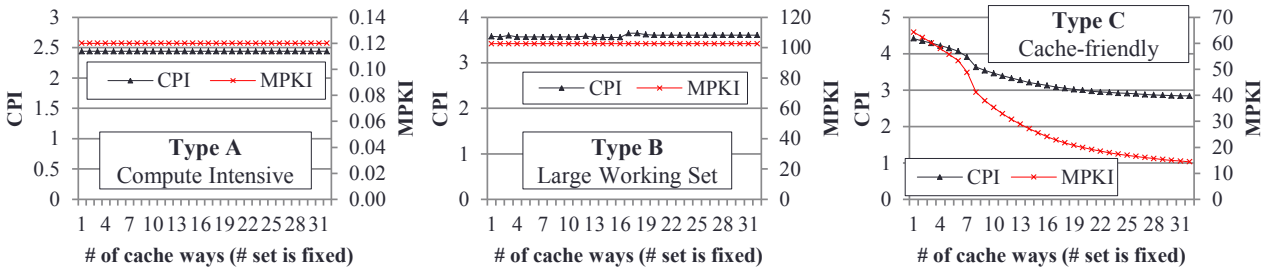


Figure 2. Application types based on how the cache affects performance (Ideal CPI is 2 in the baseline. We fix the number of cache sets (4096 sets) and vary the number of cache ways from one (256KB) to 32 (8MB)).

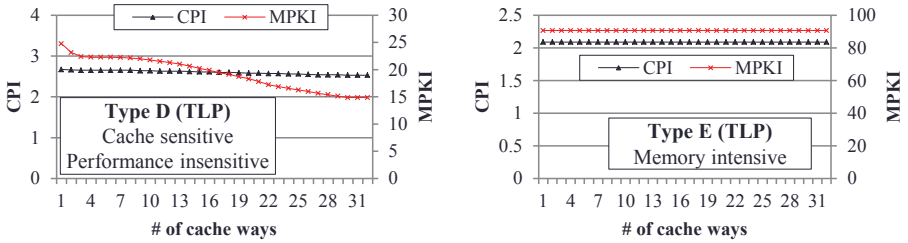


Figure 3. GPGPU unique application types.

Types D and E in Figure 3 are *unique* to GPGPU applications. These types have many cache misses, but multi-threading is so effective that *almost all memory latency can be tolerated*. We summarize these types as follows:

- Type D shows that MPKI *is reduced* as the cache size increases (cache-sensitive), but there is *little performance improvement* (performance-insensitive). In this type, multi-threading can effectively handle off-chip access latencies even without any caches, so having larger caches shows little performance improvement.
- Type E is very similar to type B. Due to the thrashing behavior, cache MPKI is very high. However, unlike type B, the observed CPI is very close to the ideal CPI of 2 since the thread-level parallelism (TLP) in type E can hide most memory latencies.

Note that types C and D are almost identical except for the change in CPI. For type C, larger caches are beneficial, but not for type D. Since these two types have identical cache behavior, we cannot differentiate them *by just checking the cache behavior*. Hence, our mechanisms aim to distinguish these two types by identifying the relationship between cache behavior and performance.

2.3. Last-Level Cache Management

Here we provide the background of previous cache mechanisms and why they may not be effective for CPU and GPGPU heterogeneous workloads. These mechanisms can be categorized into two groups, namely, *dynamic cache partitioning* and *promotion-based cache management*.

2.3.1. Dynamic Cache Partitioning

Dynamic cache partitioning mechanisms achieve their goal (throughput, fairness, bandwidth reduction, etc.) by strictly partitioning cache ways among applications. Therefore, the interference between applications can be reduced by having dedicated space for each application. Qureshi and Patt [21] proposed Utility-based Cache Partitioning (UCP), which tries to find an optimal cache partition such that the overall number of cache hits is maximized. UCP uses a set of shadow tags and hit counters to estimate the number of cache hits in each cache way for each application. Periodically, UCP runs a partitioning algorithm to calculate a new optimal partition. In every iteration of the partitioning algorithm, an application with the maximum number of hits will be chosen. The partitioning iterations continue until all ways are allocated to applications.

To minimize the adverse effect from streaming applications, Xie and Loh [28] proposed Thrasher Caging (TC). TC identifies thrashing applications by monitoring cache access frequency and the number of misses and then enforces streaming/thrashing applications to use a limited number of cache ways, called the cage. Since the antagonistic effect is isolated in the cage, TC improves performance with relatively simple thrasher classification logic.

These mechanisms are based on the assumption that high cache hit rate leads to better performance. For example, UCP [21] finds the best partition across applications that can maximize the number of overall cache hits. UCP works well when cache performance is directly correlated to the core performance, which is not always the case for GPGPU applications. They are often capable of hiding memory la-

tency with TLP (types D and E). UCP prioritizes GPGPU applications when they have a greater number of cache hits. However, this will degrade the performance of CPU applications, while there is no performance improvement on GPGPU applications. Hence, we need a new mechanism to identify the performance impact of the cache for GPGPU applications.

2.3.2. Promotion-based Cache Management

Promotion-based cache mechanisms do not strictly divide cache capacity among applications. Instead, they insert incoming blocks into a non-MRU position and promote blocks upon hits. Thus, non-temporal accesses are evicted in a short amount of time and other accesses can reside for a longer time in the cache by being promoted to the MRU position directly [9] or promoted toward the MRU position by a single position [27].

For example, the goal of Re-Reference Interval Prediction (RRIP) [9] is to be resistant to scan (non-temporal access) and thrashing (larger working set) by enforcing a shorter lifetime for each block and relying on cache block promotion upon hits.[2] The conventional LRU algorithm maintains an LRU stack for each cache set. An incoming block is inserted at the head of the stack (MRU), and the tail block (LRU) is replaced. When there is a cache hit, the hitting block will be moved to the MRU position. Thus, the lifetime of a cache block begins at the head and continues until the cache block goes through all positions in the LRU stack, which will be a waste of cache space.

On the other hand, RRIP inserts new blocks *near* the LRU position instead of at the MRU position. Upon a hit, a block is moved to the MRU position. The intuition of RRIP is to give less time for each block to stay in the cache and to give more time only to blocks with frequent reuses. Thus, RRIP can keep an active working set while minimizing the adverse effects of non-temporal accesses. RRIP also uses dynamic insertion policies to further optimize the thrashing pattern using set dueling [19].

Promotion-based cache mechanisms assume a similar number of active threads in all applications, and thereby assume a similar order of cache access rates among applications. This is a reasonable assumption when there are only CPU workloads. However, GPGPU applications have more frequent memory accesses due to having an order-of-magnitude-more threads within a core. Therefore, we have to take this different degree of access rates into account to prevent most blocks of CPU applications from being evicted by GPGPU applications even before the first promotion is performed.

[2]Note that we use a thread-aware DRRIP for our evaluations.

2.3.3. Summary of Prior Work

Table 1 summarizes how previous mechanisms work on heterogeneous workloads consisting of one CPU application and each GPGPU application type. For types A, B, D, and E, since performance is not significantly affected by the cache behavior, having fewer ways for the GPGPU application would be most beneficial. However, previous cache-oriented mechanisms favor certain applications based on the number of cache hits or cache access rate, so the GPGPU application is favored in many cases, which will degrade the performance of a CPU application.

Table 1. Application favored by mechanisms when running heterogeneous workloads (1 CPU + each type of GPGPU application).

Workloads		Favored application type			Ideal
	GPGPU	UCP	RRIP	TC	
CPU+	Type A	CPU	CPU	none	CPU
	Type B	CPU	≈ or GPGPU	CPU	CPU
	Type C	GPGPU	GPGPU	CPU	Fair share
	Type D	GPGPU	GPGPU	CPU	CPU
	Type E	CPU	≈ or GPGPU	CPU	CPU

For type C GPGPU applications, due to excessive cache accesses and a decent cache hit rate, both UCP and RRIP favor GPGPU applications. However, the ideal partitioning will be formed based on the behavior of applications, and usually, giving too much space to one application results in poor performance. On the other hand, TC can isolate most GPGPU applications by identifying them as thrashing. The *Ideal* column summarizes the ideal scenario of prioritization that maximizes system throughput.

3. TLP-Aware Cache Management Policy

This section proposes a thread-level parallelism-aware cache management policy (TAP) that consists of two components: core sampling and cache block lifetime normalization. We also propose two new TAP mechanisms: TAP-UCP and TAP-RRIP.

3.1. Core Sampling

As we discussed in Section 2, we need a new way to identify the cache-to-performance effect for GPGPU applications. Thus, we propose a sampling mechanism that applies a different policy to each core, called *core sampling*. The intuition of core sampling is that most GPGPU applications show symmetric behavior across cores on which they are running.[3] In other words, each core shows similar progress in terms of the number of retired instructions. Using this characteristic, core sampling applies a different policy to each core and periodically collects samples to see how the policies work. For example, to identify the effect of cache on performance, core sampling enforces one core (Core-POL1) to use the LRU insertion policy and another

[3]There are some exceptional cases; pipelining parallel programming patterns do not show the symmetric behavior.

core (Core-POL2) to use the MRU insertion policy. Once a period is over, the *core sampling controller* (CSC) collects the performance metrics, such as the number of retired instructions, from each core and compares them. If the CSC observes significant performance differences between Core-POL1 and Core-POL2, we can conclude that the performance of this application has been affected by the cache behavior. If the performance delta is negligible, caching is not beneficial for this application. Based on this sampling result, the CSC makes an appropriate decision in the LLC (cache insertion or partitioning) and other cores will follow this decision. Core sampling is similar to set dueling [19]. The insight of set dueling is from *Dynamic Set Sampling* (DSS) [20], which approximates the entire cache behavior by sampling a few sets in the cache with a high probability. Similarly, the symmetry in GPGPU applications makes the core sampling technique viable.

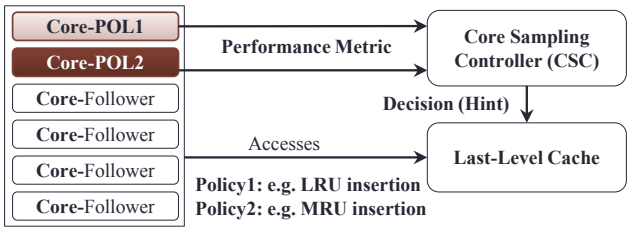


Figure 4. The core sampling framework.

Figure 4 shows the framework of core sampling. Among multiple GPU cores, one core (Core-POL1) uses policy 1, another core (Core-POL2) uses policy 2, and all others (Core-Followers) follow the decision in the LLC made by the CSC. Inputs to the CSC are performance metrics from Core-POL1 and Core-POL2.

3.1.1. Core Sampling with Cache Partitioning

When core sampling is running on top of cache partitioning, the effect of different policies for a GPGPU application is limited to its dedicated space once the partition is set for each application. For example, if a GPGPU application has only one way and CPU applications have the rest of the ways, sampling policies affect only one way for the GPGPU application. In this case, no difference exists between the MRU and LRU insertion policies. Therefore, we set core sampling to enforce Core-POL1 to bypass the LLC with cache partitioning.

3.1.2. Benchmark Classification by Core Sampling

Based on the CPI variance between Core-POL1 and Core-POL2, we categorize the variance into two groups using $Threshold_{\alpha}$. If the CPI delta is less than $Threshold_{\alpha}$, caching has little effect on performance. Thus, types A, B, D, and E can be detected. If the CPI delta is higher than $Threshold_{\alpha}$, this indicates that an application is cache-friendly. When an application has asymmetric behavior, core sampling may misidentify this application as cache-friendly. However, we found that there are only a few asymmetric benchmarks and the performance penalty of misidentifying these benchmarks is negligible. Note that we set $Threshold_{\alpha}$ to 5% from empirical data.

3.1.3. Overhead

Control logic Since we assume the logic for cache partitioning or promotion-based cache mechanisms already exists, core sampling only requires periodic logging of performance metrics from two cores and the performance-delta calculation between the two. Thus, the overhead of the control logic is almost negligible.

Storage overhead The core sampling framework requires the following additional structures. 1) *One counter per core* to count the number of retired instructions during one period: Usually, most of today's processors already have this counter. 2) *Two registers to indicate the ids of Core-POL1 and Core-POL2*: When a cache operation is performed to a cache line, the core id field is checked. If the core id matches with Core-POL1, the LRU insertion policy or LLC bypassing is used. If it matches with Core-POL2, the MRU insertion policy is used. Otherwise, the underlying mechanism will be applied to cache operations.

Table 2. Hardware complexity (our baseline has 6 GPU cores and 4 LLC tiles).

Hardware	Purpose	Overhead
20-bit counter per core	Perf. metric	20 × 6 cores = 120 bits
2 5-bit registers	Ids of Core-POL1 and Core-POL2	10-bit × 4 LLC tiles = 40 bits
Total		160 bits

Table 2 summarizes the storage overhead of core sampling. Since core sampling is applied on top of dynamic cache partitioning or promotion-based cache mechanisms, such as UCP or RRIP, we assume that the underlying hardware already supports necessary structures for them. Therefore, the overhead of core sampling is fairly negligible.

3.1.4. Discussions

Worst-performing core and load imbalance Core sampling may hurt the performance of a sampled core, Core-POL1 or Core-POL2, if a poorly performing policy is enforced during the entire execution. In set dueling, a total of 32 sets will be sampled out of 4096 sets (64B cache line, 32-way 8MB cache). Only 0.78% of the entire cache sets are affected. Since we have a much smaller number of cores than cache sets, the impact of having a poorly performing core might be significant. Also, this core may cause a load imbalance problem among cores. However, these problems can be solved by periodically rotating sampled cores instead of fixing which cores to sample.

Synchronization Most current GPUs cannot synchronize across cores, so core sampling is not affected by synchronization. However, if future GPUs support synchronization such as a barrier across cores, since all cores will make the same progress regardless of the cache policy, core sampling cannot detect performance variance between cores. In this case, we turn off core sampling and all cores follow the policy of the underlying mechanism after a few initial periods.

Handling multiple applications So far, we assume that GPUs can run only one application at a time. When a GPU core can execute more than one kernel concurrently[4], the following support is needed: (1) We need separate counters for each application to keep track of performance metrics; (2) Instead of Core-POL1 and Core-POL2 being physically fixed, the hardware can choose which core to be Core-POL1 and Core-POL2 for each application, so each application can have its own sampling information.

3.2. Cache Block Lifetime Normalization

GPGPU applications typically access caches much more frequently than CPU applications. Even though memory-intensive CPU applications also exist, the cache access rate cannot be as high as that of GPGPU applications due to a much smaller number of threads in a CPU core. Also, since GPGPU applications can maintain high throughput because of the abundant TLP in them, there will be continuous cache accesses. However, memory-intensive CPU applications cannot maintain such high throughput due to the limited TLP in them, which leads to less frequent cache accesses. As a result, there is often an order of difference in cache access frequencies between CPU and GPGPU applications. Figure 5 shows the number of memory requests per 1000 cycles (RPKC) of applications whose RPKC is in the top and bottom five, along with the median and average values from all CPU and GPGPU applications, respectively. The top five CPU applications have over 60 RPKC, but the top five GPGPU applications have over 400 and two of them have even more than 1000 RPKC. Hence, when CPU and GPGPU applications run together, we have to take into account this difference in the degree of access rates.

To solve this issue, we introduce *cache block lifetime normalization*. First, we detect access rate differences by collecting the number of cache accesses from each application. Periodically, we calculate the access ratio between applications. If the ratio exceeds the threshold, T_{xs}[5], this ratio value is stored in a 10-bit register, called *XSRATIO*. When the ratio is lower than T_{xs}, the value of XSRATIO will be set to 1. When the value of XSRATIO is greater than 1, TAP policies utilize the value of the XSRATIO register to enforce similar cache residential time to CPU and GPGPU applications. We detail how the XSRATIO register is used in the following sections.

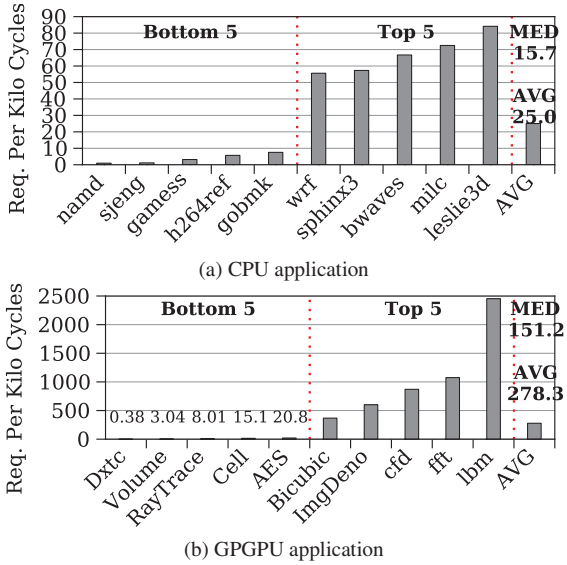


Figure 5. Memory access rate characteristics.

3.3. TAP-UCP

TAP-UCP is based on UCP [21], a dynamic cache partitioning mechanism for only CPU workloads. UCP periodically calculates an optimal partition to adapt a run-time behavior of the system. For each application, UCP maintains an LRU stack for each sampled set[6] and a hit counter for each way in all the sampled sets during a period. When a cache hit occurs in a certain position of an LRU stack, the corresponding hit counter will be incremented. Once a period is over, the partitioning algorithm iterates until all cache ways are allocated to applications. In each iteration, UCP finds the marginal utility of each application using the number of remaining ways to allocate and the hit counters of an application.[7] Then, UCP allocates one or more cache ways to the application that has the highest marginal utility.

As explained in Section 2.3.1, UCP tends to favor GPGPU applications in heterogeneous workloads. However, TAP-UCP gives more cache ways to CPU applications when core sampling identifies that a GPGPU application achieves little benefit from caching. Also, TAP-UCP adjusts the hit counters of a GPGPU application when the GPGPU application has a much greater number of cache accesses than CPU applications. To apply TAP in UCP, we need two modifications in the UCP's partitioning algorithm.

[4]NVIDIA's Fermi now supports the concurrent execution of kernels, but each core can execute only one kernel at a time.

[5]We set T_{xs} to 10, which means a GPGPU application has 10 times more accesses than the CPU application that has the highest cache access rate, via experimental results. However, we do not show these results due to space constraints.

[6]UCP collects the information only from sampled sets to reduce the overhead of maintaining an LRU stack for each set.

[7]Marginal utility is defined as the utility per unit cache resource in [21]. For more details, please refer to Algorithm 1 in Appendix A.

The first modification is that only one way is allocated for a GPGPU application when caching has little benefit on it. To implement this, we add one register to each cache, called the *UCP-Mask*. The CSC of core sampling sets the UCP-Mask register when caching is not effective; otherwise the value of the UCP-Mask remains 0. TAP-UCP checks the value of this register before performing the partitioning algorithm. When the value of the UCP-Mask is set, only CPU applications are considered for the cache way allocation.

The second modification is that when partitioning is performed, we first divide the value of the GPGPU application's hit counters by the value of the XSRATIO register, which is periodically set by cache block lifetime normalization, as described in Section 3.2. More details about the TAP-UCP partitioning algorithm can be found in Appendix A.

3.4. TAP-RRIP

First, we provide more details about the RRIP mechanism [9], which is the base of TAP-RRIP. RRIP dynamically adapts between two competing cache insertion policies, Static-RRIP (SRRIP) and Bimodal-RRIP (BRRIP), to filter out thrashing patterns. RRIP represents the insertion position as the Re-Reference Prediction Value (RRPV). With an n-bit register per cache block for the LRU counter, an RRPV of 0 indicates an MRU position and an RRPV of 2^n-1 represents an LRU position. SRRIP always inserts the incoming blocks with an RRPV of 2^n-2, which is the best performing insertion position between 0 to 2^n-1. On the other hand, BRRIP inserts blocks with an RRPV of 2^n-2 with a very small probability (5%) and for the rest, which is the majority, it places blocks with an RRPV of 2^n-1. RRIP dedicates few sets of the cache to each of the competing policies. A saturating counter, called a Policy Selector (PSEL), keeps track of which policy incurs fewer cache misses and decides the winning policy. Other non-dedicated cache sets follow the decision made by PSEL.

To apply TAP to RRIP, we need to consider two problems: 1) manage the case when a GPGPU application does not need more cache space and 2) prevent the interference by a GPGPU application with much more frequent accesses. When either or both problems exist, we enforce the BRRIP policy for the GPGPU application since BRRIP generally enforces a shorter cache lifetime than SRRIP for each block. Also, the hitting GPGPU block will not be promoted and GPGPU blocks will be replaced first when both CPU and GPGPU blocks are replaceable. In pseudo-LRU approximations including RRIP, multiple cache blocks can be in LRU positions. In this case, TAP-RRIP chooses a GPGPU block over a CPU block for the replacement.

In TAP-RRIP, we add an additional register, called the *RRIP-Mask*. The value of the RRIP-Mask register is set to 1 when 1) core sampling decides caching is not beneficial for the GPGPU application or 2) the value of the XSRATIO register is greater than 1. When the value of the RRIP-Mask register is 1, regardless of the policy decided by PSEL, the policy for the GPGPU application will be set to BRRIP. Otherwise, the winning policy by PSEL will be applied. Table 3 summarizes the policy decision of TAP-RRIP for the GPGPU application.

Table 3. TAP-RRIP policy decisions for the GPGPU application.

RRIP's decision	TAP (RRIP-Mask)	Final Policy	GPGPU type	Note
SRRIP	0	SRRIP	Type C	Base RRIP
BRRIP	0	BRRIP		
SRRIP	1	BRRIP	Types A, B, D, E	Always BRRIP
BRRIP	1	BRRIP		

4. Evaluation Methodology

4.1. Simulator

We use MacSim simulator [1] for our simulations. As the frontend, we use Pin [18] for the CPU workloads and GPUOcelot [6] for GPGPU workloads. For all simulations, we repeat early terminated applications until all other applications finish, which is a similar methodology used in [8, 9, 21, 27]. Table 4 shows the evaluated system configuration. Our baseline CPU cores are similar to the CPU cores in Intel's Sandy Bridge [7], and we model GPU cores similarly to those in NVIDIA's Fermi [14]; each core is running in SIMD fashion with multi-threading capability. The integration of both cores is illustrated in Figure 1.

Table 4. Evaluated system configurations.

CPU	1-4 cores, 3.5GHz, 4-wide, out-or-order
	gshare branch predictor
	8-way 32KB L1 I/D (2-cycle), 64B line
	8-way 256KB L2 (8-cycle), 64B line
GPU	6 cores, 1.5GHz, in-order, 2-wide 8-SIMD
	No branch predictor (switch to the next ready thread)
	8-way 32KB L1 D (2-cycle), 64B line
	4-way 4KB L1 I (1-cycle), 64B line
L3 Cache	32-way 8MB (4 tiles, 20-cycle), 64B line
NoC	20-cycle fixed latency, at most 1 req/cycle
DRAM	4 controllers, 16 banks, 4 channels
	DDR3-1333. 41.6GB/s Bandwidth, FR-FCFS

4.2. Benchmarks

Table 5 and Table 6 show the type of CPU and GPGPU applications that we use for our evaluations. We use 29 SPEC 2006 CPU benchmarks and 32 CUDA GPU benchmarks from publicly available suites, including NVIDIA CUDA SDK, Rodinia [5], Parboil [26], and ERCBench [3]. For CPU workloads, Pinpoint [17] was used to select a representative simulation region with the *ref* input set. Most GPGPU applications are run until completion.

Table 7 describes all workloads that we evaluate for heterogeneous simulation. We thoroughly evaluate our mechanisms on an excessive number of heterogeneous workloads.

Table 5. CPU benchmarks classification.

Type	Benchmarks (INT // FP)
Cache-friendly (10)	bzip2, gcc, mcf, omnetpp, astar // leslie3d, soplex, lbm, wrf, sphinx3
Streaming / Large working set (6)	libquantum // bwaves, milc, zeusmp, cactusADM, GemsFDTD
Compute intensive(13)	perlbench, gobmk, hmmer, sjeng, h264ref, xalancbmk // gamess, gromacs, namd, dealII, povray, calculix, tonto

Table 6. GPGPU benchmarks classification.

Type	Benchmarks (SDK // Rodinia // ERCBench // Parboil)
A (4)	dxtc, fastwalsh, volumerender // cell // NA // NA
B (12)	bicubic, convsep, convtex, imagedenoise, mergesort sobelfilter // hotspot, needle // sad // fft, mm, stencil
C (3)	quasirandom, sobolqrng // raytracing // NA // NA
D (4)	blackscholes, histogram, reduction // aes // NA // NA
E (9)	dct8x8, montecarlo, scalarprod // backprop, cfd, nn, bfs // sha // lbm

We form these workloads by pseudo-randomly selecting one, two, or four CPU benchmarks from cache-friendly and compute-intensive group in Table 5 and one GPGPU benchmark from each type in Table 6. For Stream-CPU workloads, in addition to streaming applications from SPEC2006 (Table 5), we add five more streaming benchmarks from the Merge [11] benchmarks.

Table 7. Heterogeneous workloads.

Type	# CPU	# GPGPU	# of total workloads
1-CPU	1	1	152 workloads
2-CPU	2	1	150 workloads
4-CPU	4	1	75 workloads
Stream-CPU	1	1	25 workloads

4.3. Evaluation Metric

We use the geometric mean (Eq. 1) of the speedup of each application (Eq. 2) as the main evaluation metric.

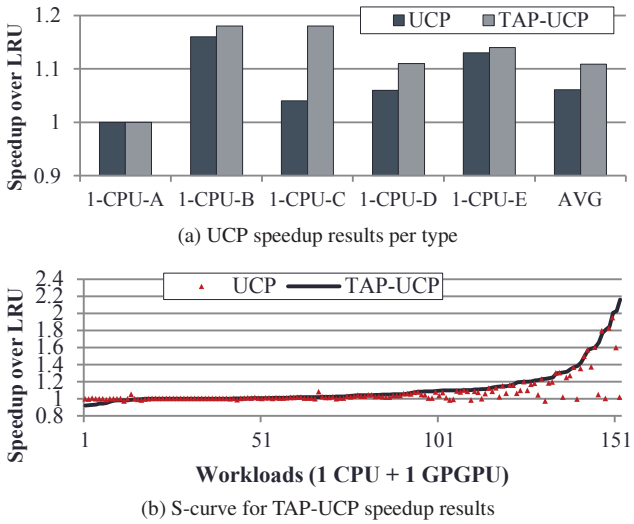
$$speedup = geomean(speedup_{(0\ to\ n-1)}) \quad (1)$$

$$speedup_i = \frac{IPC_i}{IPC_i^{baseline}} \quad (2)$$

5. Experimental Evaluation

5.1. TAP-UCP Evaluation

Figure 6 shows the base UCP and TAP-UCP speedup results normalized to the LRU replacement policy on all 1-CPU workloads (one CPU + one GPGPU). Figure 6 (a) shows the performance results for each GPGPU application type. For 1-CPU-A[8], 1-CPU-B, and 1-CPU-E workloads, as explained in Table 1, since these types of GPGPU applications do not have many cache hits, UCP can successfully partition cache space toward being CPU-friendly. Therefore, the base UCP performs well on these workloads and improves performance over LRU by 0%, 15%, and 12% for workloads 1-CPU-A, 1-CPU-B, and 1-CPU-E, respectively. Note that since type A applications are computation-intensive, even LRU works well.



(a) UCP speedup results per type

(b) S-curve for TAP-UCP speedup results

Figure 6. TAP-UCP speedup results.

For 1-CPU-C and 1-CPU-D, UCP is less effective than other types. For 1-CPU-C, we observe that the number of cache accesses and cache hits of GPGPU applications is much higher than that of CPUs (at least an order). As a result, UCP strongly favors the GPGPU application, so there is a severe performance degradation in the CPU application. Therefore, UCP shows only a 3% improvement over LRU. However, by considering the different access rates in two workloads, TAP-UCP successfully balances cache space between CPU and GPGPU applications. TAP-UCP shows performance improvements of 14% and 17% compared to UCP and LRU, respectively, for 1-CPU-C workloads. For 1-CPU-D, although larger caches are not beneficial for GPGPU applications since they have more cache hits than CPU applications, UCP naturally favors the GPGPU applications. However, the cache hit pattern of the GPGPU applications often shows a strong locality near the MRU position, so UCP stops the allocation for GPGPU applications after a few hot cache ways. As a result, UCP performs better than LRU by 5% on average. The performance of TAP-UCP is 5% better than UCP by detecting when more caching is not beneficial for GPGPU applications.

From the s-curve[9] result in Figure 6 (b), TAP-UCP usu-

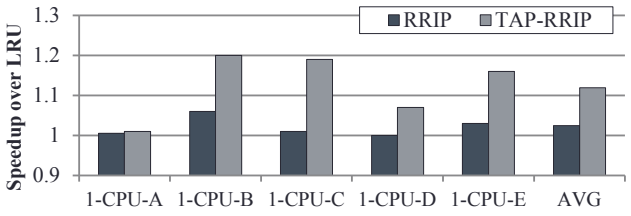
[8]CPU application with one type A GPGPU application. Same rule applies to other types.

[9]For all s-curve figures from now on, we sort all results by the performance of the TAP mechanisms in ascending order.

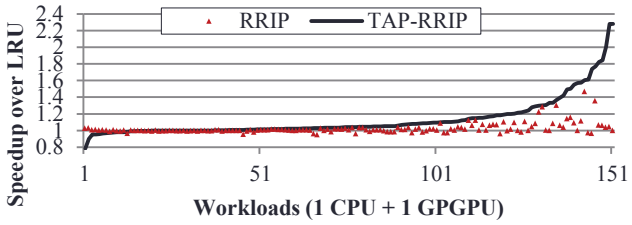
ally outperforms UCP except in a few cases with type C GPGPU applications. When a CPU application is running with a type C GPGPU application, giving very few ways to GPGPU applications increases the bandwidth requirement significantly. As a result, the average off-chip access latency increases, so the performance of all other memory-intensive benchmarks is degraded severely. We see these cases only in seven workloads out of 152. In our future work, we will monitor bandwidth increases to prevent these negative cases. Overall, UCP performs 6% better than LRU, and TAP-UCP improves UCP by 5% and LRU by 11% across 152 heterogeneous workloads.

5.2. TAP-RRIP Evaluation

Figure 7 (a) presents the speedup results of RRIP and TAP-RRIP for each GPGPU type. We use a thread-aware DRRIP, which is denoted as RRIP in the figures, for evaluations with a 2-bit register for each cache block. Other configurations are the same as in [9]. The base RRIP performs similarly to LRU. As explained in Section 2.3.2, RRIP favors GPGPU applications because of its more frequent cache accesses. Thus, GPGPU blocks occupy the majority of cache space. On the other hand, TAP-RRIP tries to give less space to GPGPU blocks if core sampling identifies that more caching is not beneficial.



(a) RRIP speedup results per type



(b) S-curve for TAP-RRIP speedup results

Figure 7. TAP-RRIP speedup results.

Figure 7 (b) shows the s-curve for the performance on all 152 workloads. Although RRIP does not show many cases with degradation, RRIP is not usually effective and performs similarly to LRU. However, TAP-RRIP shows performance improvement in more than half of the evaluated workloads. Two TAP-RRIP cases show degradation of more than 5%. Again, this is the problem due to type C GPGPU applications (too little space is given to the GPGPU application, so the bandwidth is saturated).

On average, the base RRIP performs better than LRU by 3% while TAP-RRIP improves the performance of RRIP and LRU by 9% and 12%, respectively.

5.3. Streaming CPU Application

When a streaming CPU application is running with a GPGPU application, our TAP mechanisms tend to unnecessarily penalize GPGPU applications even though the streaming CPU application does not need any cache space. Since we have only considered the adverse effect of GPGPU applications, the basic TAP mechanisms cannot effectively handle this case. Thus, we add a streaming behavior detection mechanism similar to [27, 28], which requires only a few counters. Then, we minimize space usage by CPU applications once they are identified as streaming. The enhanced TAP-UCP will allocate only one way to a streaming CPU application and the enhanced TAP-RRIP will reset the value of the RRIP-Mask register to operate as the base RRIP, which works well for streaming CPU applications.

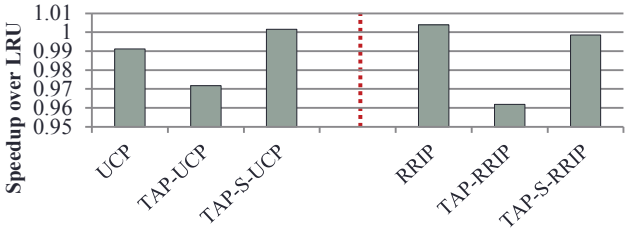


Figure 8. Enhanced TAP mechanism (TAP-S) results.

Figure 8 shows the performance results of the enhanced TAP mechanisms (TAP-S) that consider the streaming behavior of CPU applications on the 25 Stream-CPU workloads in Table 7. The basic TAP mechanisms degrade performance by 3% and 4% over LRU, respectively. However, the TAP-S mechanisms solve the problem of basic mechanisms and show a performance similar to LRU. Since all other previous mechanisms, including LRU, can handle the streaming application correctly, the TAP mechanisms cannot gain further benefit. Note that the TAP-S mechanisms do not change the performance of other workloads.

5.4. Multiple CPU Applications

So far, we have evaluated the combinations of one CPU and one GPGPU application. In this section, we evaluate multiple CPU applications running with one GPGPU application (2-CPU and 4-CPU workloads in Table 7). As the number of concurrently running applications increases, the interference by other applications will also increase. Thus, the role of intelligent cache management becomes more crucial. Figure 9 shows evaluations on 150 2-CPU and 75 4-CPU workloads. TAP-UCP shows up to a 2.33 times and 1.93 times speedup on 2-CPU and 4-CPU workloads, re-

spectively.[10] TAP-UCP performs usually no worse than the base UCP except in a few cases. In this case, two or four memory-intensive CPU benchmarks are running with one type C GPGPU application. On average, TAP-UCP improves the performance of LRU by 12.5% and 17.4% on 2-CPU and 4-CPU workloads, respectively, while UCP improves by 7.6% and 6.1%.

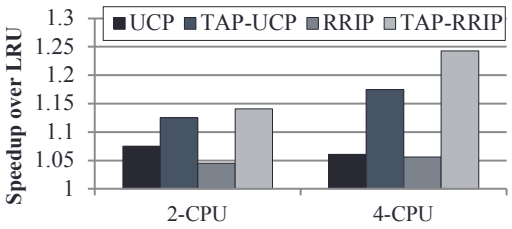


Figure 9. Multiple CPU application results.

TAP-RRIP shows up to a 2.3 times and 2.2 times speedup on 2-CPU and 4-CPU workloads, respectively. On average, RRIP improves the performance of LRU by 4.5% and 5.6% on 2-CPU and 4-CPU workloads, respectively, while TAP-RRIP improves even more, by 14.1% and 24.3%. From multi-CPU evaluations of the TAP mechanisms, we conclude that our TAP mechanisms show good scalability by intelligently handling inter-application interference.

5.5. Comparison to Static Partitioning

Instead of using dynamic cache partitioning, a cache architecture can be statically partitioned between CPUs and GPUs, but statically partitioned caches cannot use the resources efficiently. In other words, it cannot adapt to workload characteristics at run-time. In this section, we evaluate a system that statically partitions the LLC between the CPUs and GPUs evenly. All CPU cores (at most 4) share 16 ways of the LLC regardless of the number of concurrently running CPU applications, and the GPU cores (6 cores) share the rest of the 16 ways.

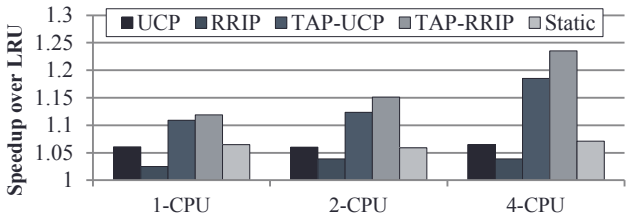


Figure 10. Static partitioning results.

Figure 10 shows the TAP results compared to static partitioning. For 1-CPU workloads, static partitioning shows a 6.5% improvement over LRU, while TAP-UCP and TAP-RRIP show 11% and 12% improvements. However, as the number of concurrently running applications increases, static partitioning does not show further improvement (7% over in both 2-CPU and 4-CPU workloads), while the benefit of the TAP mechanisms continuously increases (TAP-UCP: 12% and 19%, TAP-RRIP: 15% and 24% for 2-CPU and 4-CPU workloads, respectively). Moreover, static partitioning performs slightly worse than LRU in many cases (52 out of 152 in 1-CPU, 54 out of 150 in 2-CPU, and 28 out of 75 in 4-CPU workloads, respectively), even though the average is 7% better than that of LRU.[11] We conclude that static partitioning on average performs better than LRU, but it cannot adapt to workload characteristics, especially when the number of applications increases.

5.6. Cache Sensitivity Evaluation

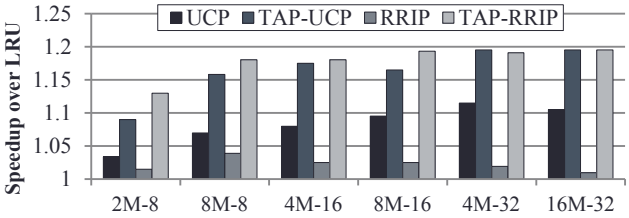


Figure 11. Cache sensitivity results (size-associativity).

Figure 11 shows the performance results with other cache configurations. We vary the associativity and size of caches. As shown, our TAP mechanisms constantly outperform their corresponding mechanisms, UCP and RRIP, in all configurations. Therefore, we can conclude that our TAP mechanisms are robust to cache configurations.

5.7. Comparison to Other Mechanisms

In this section, we compare the TAP mechanisms with other cache mechanisms, including TADIP [8], PIPP [27], and TC [28] along with UCP and RRIP. TADIP is a dynamic insertion policy (DIP) that dynamically identifies the application characteristic and inserts single-use blocks (dead on fill) in the LRU position to evict as early as possible. PIPP [27] pseudo partitions cache space to each application by having a different insert position for each application, which is determined using a utility monitor as in UCP. Upon hits, each block is promoted toward the MRU by one position. PIPP also considers the streaming behavior of an application. When an application shows streaming behavior, PIPP assigns only one way and allows promotion with a very small probability (1/128).

Figure 12 shows the speedup results. As explained in Section 2.3.2, if cache space is not strictly partitioned, an application that has more frequent cache accesses is favored. As a result, TADIP also favors GPGPU applications, thereby showing only 3% improvement over LRU. On the other hand, PIPP can be effective by handling GPGPU applications as streaming. Since most GPGPU applications

[10] The detailed data is not shown due to space constraints.

[11] Because of the space limitation, we cannot present the detailed data.

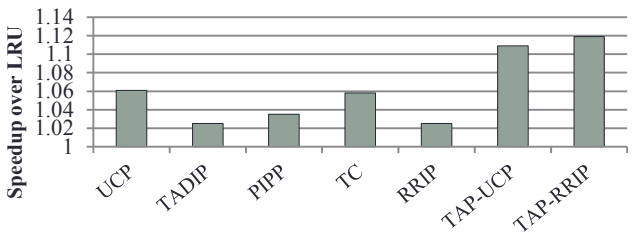


Figure 12. TAP comparison to other policies.

are identified as streaming, PIPP can be effective for types A, B, D, and E GPGPU applications. However, for type C, due to the saturated bandwidth from off-chip accesses by GPGPU applications, PIPP is not as effective as it is for other types. TC has similar benefits and problems as PIPP. On average, PIPP and TC improve performance by 4% and 6%, respectively, over LRU. Our TAP mechanisms outperform these previous mechanisms by exploiting GPGPU-specific characteristics.

6. Related Work

Dynamic cache partitioning: Suh et al. [23, 24] first proposed dynamic cache partitioning schemes in chip multiprocessors that consider the cache utility (number of cache hits) using a set of in-cache counters to estimate the cache-miss rate as a function of cache size. However, since the utility information is acquired within a cache, information for an application cannot be isolated from other applications' intervention.

Kim et al. [10] considered the fairness problem from cache sharing such that slowdown due to the cache sharing is uniform to all applications. Moretó et al. [13] proposed MLP-aware cache partitioning, where the number of overlapped misses will decide the priority of each cache miss, so misses with less MLP will have a higher priority. IPC-based cache partitioning [25] considered performance as the miss rate varies. Even though the cache-miss rate is strongly related to performance, it does not always match the performance. However, since the baseline performance model is again based on the miss rate and its penalty, it cannot distinguish GPGPU-specific characteristics. Yu and Petrov [29] considered bandwidth reduction through cache partitioning. Srikantaiah et al. proposed the SHARP control architecture [22] to provide QoS while achieving good cache space utilization. Based on the cache performance model, each application estimates the cache requirement and central controllers collect this information and coordinate requirements from all applications. Liu et al. [12] considered an off-chip bandwidth partitioning mechanism on top of cache partitioning mechanisms.

LLC policies by application level management: We have already compared TADIP [8] and PIPP [27] in Section 5.7. Pseudo-LIFO [4] mechanisms are a new family of replacement policies based on the fill stack rather than the recency stack of the LRU. The intuition of pseudo-LIFO is that most hits are from the top of the fill stack and the remaining hits are usually from the lower part of the stack. Pseudo-LIFO exploits this behavior by replacing blocks in the upper part of the stack, which are likely to be unused.

7. Conclusion

LLC management is an important problem in today's chip multi-processors and in future many-core-heterogeneous processors. Many researchers have proposed various mechanisms for throughput, fairness, or bandwidth. However, none of the previous mechanisms consider GPGPU-specific characteristics in heterogeneous workloads such as underlying massive multi-threading and the different degree of access rates between CPU and GPGPU applications. Therefore, when CPU applications are running with a GPGPU application, the previous mechanisms will not deliver the expected outcome and may even perform worse than the LRU replacement policy. In order to identify the characteristics of a GPGPU application, we propose core sampling, which is a simple yet effective technique to profile a GPGPU application at run-time. By applying core sampling to UCP and RRIP and considering the different degree of access rates, we propose the TAP-UCP and TAP-RRIP mechanisms. We evaluate the TAP mechanisms on 152 heterogeneous workloads and show that they improve the performance by 5% and 10% compared to UCP and RRIP and 11% and 12% to LRU. In future work, we will consider bandwidth effects in shared cache management on heterogeneous architectures.

Acknowledgments

Many thanks to Chang Joo Lee, Aamer Jaleel, Moinuddin Qureshi, Yuejian Xie, Nagesh B. Lakshminarayana, Jae Woong Sim, Puyan Lofti, other HPArch members, and the anonymous reviewers for their suggestions and feedback on improving the paper. We gratefully acknowledge the support of the National Science Foundation (NSF) CAREER award 1139083, the U.S. Department of Energy including Sandia National Laboratories, Intel Corporation, Advanced Micro Devices, Microsoft Research, and the equipment donations from NVIDIA.

References

[1] MacSim. http://code.google.com/p/macsim/.
[2] AMD. Fusion. http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx.
[3] D. Chang, C. Jenkins, P. Garcia, S. Gilani, P. Aguilera, A. Nagarajan, M. Anderson, M. Kenny, S. Bauer, M. Schulte, and K. Compton. ERCBench: An open-source benchmark suite for embedded and reconfigurable computing. In *FPL'10*, pages 408–413, 2010.
[4] M. Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *MICRO-42*, pages 401–412, 2009.

[5] 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'09*, 2009.
[6] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *PACT-19*, 2010.
[7] Intel. Sandy Bridge. http://software.intel.com/en-us/articles/sandy-bridge/.
[8] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *PACT-17*, pages 208–219, 2008.
[9] 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.
[10] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT-13*, pages 111–122, 2004.
[11] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII*, 2008.
[12] F. Liu, X. Jiang, and Y. Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *HPCA-16*, pages 1 –12, jan. 2010.
[13] M. Moretó, F. J. Cazorla, A. Ramírez, and M. Valero. MLP-aware dynamic cache partitioning. In *HiPEAC'08*, volume 4917, pages 337–352. Springer, 2008.
[14] NVIDIA. Fermi: Nvidia's next generation cuda compute architecture. http://www.nvidia.com/fermi.
[15] NVIDIA. Project denver. http://blogs.nvidia.com/2011/01/project-denver-processor-to-usher-in-new-era-of-computing/.
[16] OpenCL. The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl.
[17] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *MICRO-37*, pages 81–92, 2004.
[18] Pin. *A Binary Instrumentation Tool*. http://www.pintool.org.
[19] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA-29*, pages 381–391, 2007.
[20] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA-28*, pages 167–178, 2006.
[21] 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.
[22] S. Srikantaiah, M. Kandemir, and Q. Wang. Sharp control: Controlled shared cache management in chip multiprocessors. In *MICRO-42*, pages 517 –528, dec. 2009.
[23] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA-8*, pages 117–128, feb. 2002.
[24] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.
[25] G. Suo, X. Yang, G. Liu, J. Wu, K. Zeng, B. Zhang, and Y. Lin. IPC-based cache partitioning: An ipc-oriented dynamic shared cache partitioning mechanism. In *ICHIT'08*, pages 399 –406, aug. 2008.
[26] The IMPACT Research Group, UIUC. Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.
[27] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA-31*, pages 174–183, 2009.
[28] Y. Xie and G. H. Loh. Scalable shared-cache management by containing thrashing workloads. In Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, editors, *HiPEAC'10*, volume 5952, pages 262–276. Springer, 2010.
[29] C. Yu and P. Petrov. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *DAC'10*, pages 132–137, 2010.

Appendix

A. TAP-UCP

In this section, we provide the partitioning algorithm of the TAP-UCP mechanism in Algorithm 1. Based on the UCP's look-ahead partitioning algorithm [21], we extend it by applying core sampling (line 8:11) and cache block lifetime normalization (line 3:5).

Algorithm 1 TAP-UCP algorithm (modified UCP)

```
 1: balance = N
 2: allocation[i] = 0 for each competing application i
 3: if XSRATIO > 1    // TAP-UCP begin .....................
 4:     foreach way j in GPGPU application i do
 5:         way_counter_i[j] /= XSRATIO    // TAP-UCP end .....
 6: while balance do:
 7:     foreach application i do:
 8:         // TAP-UCP begin ..................................
 9:         if application i is GPGPU application and UCP-Mask == 1
10:             continue
11:         // TAP-UCP end ....................................
12:         alloc = allocations[i]
13:         max_mu[i] = get_max_mu(i, alloc, balance)
14:         blocks_req[i] = min blocks to get max_mu[i] for i
15:     winner = application with maximum value of max_mu
16:     allocations[winner] += blocks_req[winner]
17:     balance -= blocks_req[winner]
18: return allocations
19:
20: // get the maximum marginal utility of an application
21: get_max_mu(app, alloc, balance):
22:     max_mu = 0
23:     for (ii=1;ii<=balance;ii++) do:
24:         mu = get_mu_value(p, alloc, alloc+ii)
25:         if (mu > max_mu) max_mu = mu
26:     return max_mu
27:
28: // get a marginal utility
29: get_mu_value(app, a, b):
30:     U = change in misses for application p when the number of blocks
31:         assigned to it increases from a-way to b-way (a < b)
32:     return U/(b-a)
```