00

# Performance-Energy Considerations for Shared Cache Management in a Heterogeneous Multicore Processor

Anup Holey†, Intel Corporation
Vineeth Mekkat†, Intel Corporation
Pen-Chung Yew, University of Minnesota - Twin Cities
Antonia Zhai, University of Minnesota - Twin Cities

Heterogeneous multicore processors that integrate CPU cores and data-parallel accelerators such as GPU cores onto the same die raise several new issues for sharing various on-chip resources. The shared last-level cache (LLC) is one of the most important shared resources due to its impact on performance. Accesses to the shared LLC in heterogeneous multicore processors can be dominated by the GPU due to the significantly higher number of concurrent threads supported by the architecture. Under current cache management policies, the CPU applications' share of the LLC can be significantly reduced in the presence of competing GPU applications. For many CPU applications, a reduced share of the LLC could lead to significant performance degradation. On the contrary, GPU applications can tolerate increase in memory access latency when there is sufficient thread-level parallelism. In addition to the performance challenge, introduction of diverse cores on to the same die changes the energy consumption profile and, in turn, affects the energy efficiency of the processor.

In this work, we propose Heterogeneous LLC Management (HeLM), a novel shared LLC management policy that takes advantage of the GPU's tolerance for memory access latency. HeLM is able to throttle GPU LLC accesses and yield LLC space to cache sensitive CPU applications. This throttling is achieved by allowing GPU accesses to bypass the LLC when an increase in memory access latency can be tolerated. The latency tolerance of a GPU application is determined by the availability of thread-level parallelism, which is measured at runtime as the average number of threads that are available for issuing. For a baseline configuration with two CPU cores and four GPU cores, modelled after existing heterogeneous processor designs, HeLM outperforms LRU policy by 10.4%. Additionally, HeLM also outperforms competing policies. Our evaluations show that HeLM is able to sustain performance with varying core mix.

In addition to the performance benefit, bypassing also reduces total accesses to the LLC leading to a reduction in the energy consumption of the LLC module. However, LLC bypassing has the potential to increase off-chip bandwidth utilization and DRAM energy consumption. Our experiments show that HeLM exhibits better energy efficiency by reducing $ED^2$ value by 18% over LRU, while impacting only a 7% increase in off-chip bandwidth utilization.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache Memories*; C.1.3 [**Computer Systems Organization**]: Processor Architectures—*Heterogeneous (Hybrid) Systems*

General Terms: Architecture, Experimentation, Performance

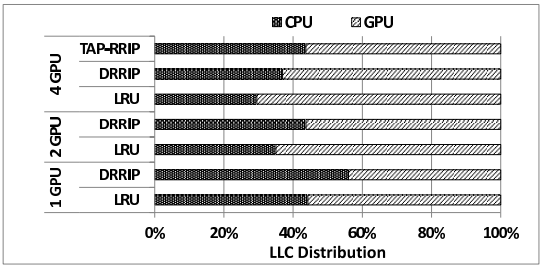
Additional Key Words and Phrases: Heterogeneous Multicore, Cache Management Policy, Last-level Cache, Bypassing

**ACM Reference Format:**
Anup Holey, Vineeth Mekkat, Pen-Chung Yew, and Antonia Zhai. 2014. Performance-Energy Considerations in Shared Cache Management in a Heterogeneous Multicore Processor. *ACM Trans. Architec. Code Optim.* 0, 0, Article 00 ( 201X), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

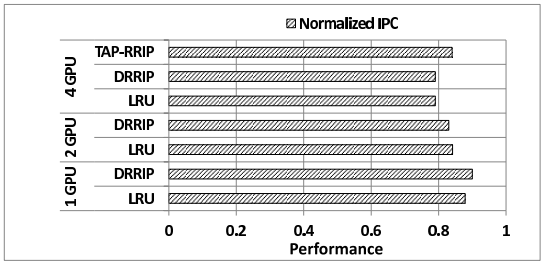
---

†Authors were affiliated to the University of Minnesota when this work was done.
This work is supported in part by National Science Foundation grants CCF-0916583 and CPS-0931931.
Author's addresses: A. Holey, Intel Corporation, 1900 Prairie City Road, Folsom, CA 95630; email: anup.holey@intel.com. V. Mekkat, Intel Corporation, 3600 Juliette Lane, Santa Clara, CA 95054; email: vineeth.mekkat@intel.com. P.-C. Yew and A. Zhai, Department of Computer Science and Engineering, University of Minnesota, 200 Union Street, Keller Hall 4-192, Minneapolis, MN 55455; email: {yew, zhai}@cs.umn.edu.

DOI: http://dx.doi.org/10.1145/0000000.0000000




(a) Cache occupancy of the CPU and the GPU cores. Occupancy refers to the distribution of the LLC space between applications.




(b) Normalized IPC for cache sensitive CPU application (401.bzip2) when GPU is introduced. IPC is normalized to the IPC of 401.bzip2 executing on the heterogeneous processor without interference from the GPU cores.

Fig. 1: The performance impact on a cache sensitive CPU application sharing the LLC with GPU application under various cache replacement policies. Cache sensitive SPEC [Spradling 2007] application 401.bzip2 executes on the CPU core. The performance impact is measured across the set of GPU benchmarks shown in Table III. Three configurations with varying GPU core counts are evaluated. TAP-RRIP [Lee and Kim 2012] results are shown only for 4 GPU configuration as TAP-RRIP needs more than two GPU cores for full functioning.

## 1. INTRODUCTION

Advances in semiconductor technology and the urgent need for energy efficient computation have facilitated the integration of computational cores that are heterogeneous in nature onto the same die. Data-parallel accelerators such as Graphic Processing Units (GPU) are among the most popular accelerator cores used in such designs. With easy to adopt programming models, such as Nvidia CUDA [NVIDIA Corporation 2007] and OpenCL [Khronos Group 2009], these data-parallel cores are now being employed to accelerate diverse workloads. Availability of heterogeneous multicore systems such as AMD Fusion [Brookwood 2010] and Intel Sandy Bridge [Intel Corporation 2009] suggests that multicore designs with heterogeneous processing elements are becoming part of mainstream computing. Diversity in the performance characteristics of these computational cores presents a unique set of challenges in designing these heterogeneous multicore processors.

In heterogeneous multicore systems, the efficient sharing of on-chip resources such as the last-level cache (LLC) is key to performance. However, the integration of CPU and GPU cores onto the same die leads to competition in the LLC that does not exist in homogeneous systems. First, the difference in cache sensitivity among diverse cores imply difference in performance benefits obtained from owning the same amount of cache space. Second, GPU cores with a large number of threads can potentially dominate accesses to the LLC, and consequently, skew existing cache sharing policies in favor of the GPU cores. As a result, GPU cores occupy an unfair share of the LLC with existing policies.

Figure 1 shows the performance of various cache replacement policies in a heterogeneous execution environment where 401.bzip2 (from the SPEC CPU2006 benchmark suite [Spradling 2007]) executing on a single CPU core shares a 2MB LLC with a GPU benchmark (from the AMD APP benchmark suite [Advanced Micro Devices Incorporated 2011]) executing on the GPU. The applications are listed in Table III and the details of the experiment and processor configurations are provided in Section 4. Figure 1(a) shows the average LLC occupancy and Figure 1(b) shows the normalized IPC of the CPU application across all the GPU benchmarks. Occupancy refers to the distribution of the LLC space between applications. Since 401.bzip2 is cache sensitive, while most of the GPU applications are not, it is desirable to allocate a larger share of the LLC to the CPU application. However, for the basic Least Recently Used (LRU) policy, we observe that a major portion of the LLC is occupied by the GPU application. This leads to significant performance degradation for the CPU application under the LRU policy as shown in Figure 1(b).

Prior works have shown that judicious sharing of the LLC can improve the overall performance when diverse workloads share homogeneous multicore systems [Suh et al. 2004; Moreto et al. 2008; Kim et al.

2004; Qureshi and Patt 2006; Xie and Loh 2010; Qureshi et al. 2007; Jaleel et al. 2010; Xie and Loh 2009]. To evaluate whether these techniques can be adopted by heterogeneous multicore processors, we study several recently proposed policies. Dynamic Re-Reference Interval Prediction (DRRIP) [Jaleel et al. 2010] is a cache management policy developed primarily for homogeneous multicore processors. DRRIP predicts whether the re-reference (reuse) interval of cache lines are *intermediate* or *distant*, and inserts lines at non-MRU (Most Recently Used) position based on the prediction. If a line is re-used after insertion into the LLC, it is promoted by increasing its age to improve its lifetime in the cache. Non-MRU insertion of cache lines performs better than MRU insertion because most of the lines do not exhibit immediate re-reference. Figure 1(a) indicates that DRRIP provides little improvement in LLC occupancy in a heterogeneous environment as the policy is overwhelmed by an order of magnitude difference between the memory access rates of the CPU and the GPU cores. The performance impact of the unbalanced LLC occupancy is shown in Figure 1(b).

We are aware of only one existing work, TLP-Aware Cache Management Policy (TAP) [Lee and Kim 2012], that addresses the diversity of on-chip cores while designing the LLC sharing policy. TAP identifies the cache sensitivity of the GPU application, and the difference in LLC access rates between the CPU and GPU cores. This information is used to influence the decisions made by the underlying cache management policy. When these metrics indicate a cache sensitive GPU application, both CPU and GPU cores are given equal priority. On the other hand, if GPU application is cache insensitive, the GPU core is given a lower priority by the underlying policy.

TAP, although designed for heterogeneous multicore processors, still allocates a large portion of the cache to the cache-insensitive GPU application. Consequently, the performance degradation due to LLC sharing is still significant for the cache sensitive CPU application as shown in Figure 1. Several reasons prohibit TAP from achieving the desired performance. First, the *core sampling* technique used in TAP to measure the cache sensitivity of the GPU application leaves a significant amount of GPU dead-blocks in the LLC. Second, TAP takes the same decision for all GPU memory accesses in a sampling period, and is slow to adapt to the runtime variations in the application's behavior. A more fine-grained control over the GPU LLC share could potentially improve the utilization of the shared LLC. We discuss TAP in detail in Section 5.3.3.

In addition to the performance aspect, the presence of diverse cores could change the energy consumption profile, both on-chip as well as off-chip, for the heterogeneous multicore processor under existing policies. This could result in a significant increase in energy consumption at the LLC module if the order of magnitude higher access rate of GPU is not efficiently handled by the cache replacement policy. Also, since this increase in energy consumption does not imply performance improvement in a typical cache insensitive GPU application, the energy efficiency of the processor is impacted. Bandwidth utilization is another characteristic that would also be significantly impacted by the high memory access rate of GPU cores. Since energy consumption and bandwidth utilization have already turned into first-order constraints in processor design, these aspects could be significant challenge to the viability of cache management policies in future processor designs.

To handle these challenges, performance as well as energy efficiency, we study the characteristics of the GPU architecture. The GPU core can support thousands of active threads simultaneously. Thus, the thread-level parallelism (TLP) available with the GPU core is orders of magnitude higher than that with the CPU core. This higher level of TLP aids the GPU core in tolerating longer memory access latency by scheduling threads that are ready to execute. Our experiments show that majority of the GPU applications we study have a high level of memory access latency tolerance. Taking into consideration the latency tolerance of the GPU, we propose Heterogeneous LLC Management (HeLM), a mechanism for managing shared LLC in heterogeneous multicore processors [Mekkat et al. 2013].

HeLM is a cache replacement policy that improves the effectiveness of the shared LLC in a heterogeneous environment by utilizing the available TLP in GPU applications. Under the HeLM policy, GPU LLC accesses are throttled by allowing memory accesses to selectively bypass the LLC; and consequently, the cache sensitive CPU application is able to utilize a larger portion of the cache. Our evaluations show that HeLM is able to improve the overall performance of heterogeneous workloads significantly. We also conduct

a detailed study on the energy consumption and the energy efficiency of HeLM to evaluate the feasibility of using HeLM on real systems. Our evaluations show that HeLM outperforms other policies in energy efficiency.

Overall, the contributions in this work are as follows:

— We analyze GPU application characteristics and identify available TLP as an efficient runtime metric to measure the memory access latency tolerance of the GPU application. We also find that LLC bypassing provides sufficient aggressiveness in managing shared LLC in heterogeneous multicore processors.
— We propose HeLM, a runtime mechanism, that dynamically determines the cache sensitivity of both CPU and GPU applications, and adapts the cache management policy based on this information.
— We design, implement, and evaluate HeLM, and demonstrate that HeLM is able to improve the performance as well as energy efficiency of shared cache management in a heterogeneous multicore processor.

The rest of this paper is organized as follows. Section 2 explores the challenges in LLC sharing in a heterogeneous environment, based on which, Section 3 describes the architecture of HeLM. Our evaluation methodology is described in Section 4. HeLM's performance is evaluated in Section 5 and energy efficiency in Section 6. We discuss related works in Section 7 and conclude in Section 8.

## 2. CHALLENGES & OPPORTUNITIES

The heterogeneous multicore architecture we address in this work is depicted in Figure 2. This design is modelled after AMD Fusion APU [Brookwood 2010]. The processor consists of several CPU and GPU cores each with its own private cache. These cores share the LLC and DRAM controllers, and the modules communicate through an on-chip interconnection network. Efficient sharing of on-chip resources is critical to the performance of a multicore processor. The last-level cache is one of the most important among these resources.

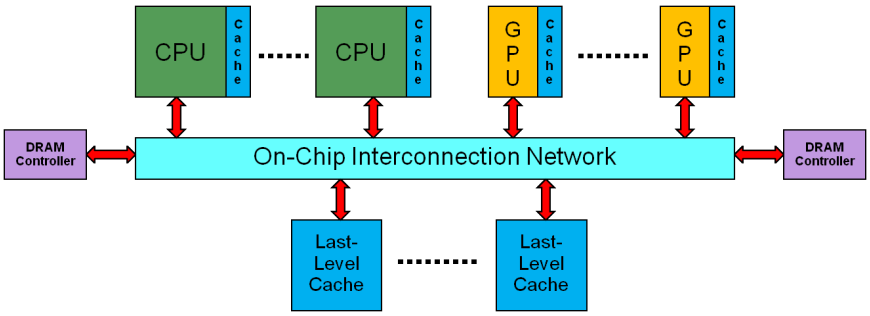



Fig. 2: A heterogeneous multicore processor with CPU and GPU cores sharing the LLC.

Existing cache management policies, developed for homogeneous multicore processors, face difficulty in adapting to heterogeneous architectures. These mechanisms [Suh et al. 2004; Moreto et al. 2008; Kim et al. 2004; Qureshi and Patt 2006; Xie and Loh 2010; Qureshi et al. 2007; Jaleel et al. 2010; Xie and Loh 2009], that were proposed for homogeneous multicore processors, do not consider the diversity of core characteristics in their design. While many mechanisms [Qureshi and Patt 2006; Xie and Loh 2010; Qureshi et al. 2007; Jaleel et al. 2010; Xie and Loh 2009] consider the cache sensitivity of the application, they do not consider the difference in LLC access rate between the diverse cores in a heterogeneous multicore. An order of magnitude higher access rate from the GPU cores, compared to the CPU cores, tends to skew their judgement in favor of the GPU.

### 2.1. Challenges in LLC sharing

Sharing of the LLC among cores in a heterogeneous multicore processor introduces several challenges. This section presents the challenges in determining the cache sensitivity of various applications executing on individual cores; and devising cache management mechanisms that can cope with cores having widely divergent memory access patterns.

Cache sensitivity indicates how much the performance of an application can benefit from an increase in cache capacity. Cache management policies can utilize cache sensitivity as a metric to determine how to best share cache capacity between cores. In CPU-based homogeneous multicore systems, this issue has been studied extensively. Techniques, such as *set dueling* [Qureshi et al. 2007], have been demonstrated effective in improving cache utilization. It is worth pointing out that, in such CPU-based systems [Qureshi and Patt 2006; Xie and Loh 2010; Qureshi et al. 2007; Jaleel et al. 2010; Xie and Loh 2009], cache sensitivity is often measured in terms of variations in cache miss rates across different cores as cache capacity allocation varies [Qureshi and Patt 2006; Xie and Loh 2010] or as cache replacement policy changes [Qureshi et al. 2007; Jaleel et al. 2010; Xie and Loh 2009].

While for CPU cores, change in cache miss rate is a direct indicator of cache sensitivity, in GPU cores, increase in cache miss rate does not necessarily lead to performance degradation. Figure 3 shows the cache sensitivity of a GPU application, BoxFilter [Advanced Micro Devices Incorporated 2011], where cache miss rate does not translate directly into performance. This is because GPU cores can tolerate memory access latency by context switching between a large number of concurrently active threads. Thus, cache miss rate is not a good indicator of the cache sensitivity of the GPU. GPU-specific techniques must be developed to determine the cache sensitivity of GPU workloads.

There are two classes of techniques for managing the shared caches: i) partitioning the cache ways among applications; and ii) prioritizing the insertion/eviction of blocks from different applications. When workloads with differing cache sensitivities share a cache, one of these techniques could be employed to enhance cache utilization and maximize the overall performance. However, when GPU workload is sharing the cache with CPU, previously proposed mechanisms are often unable to make judicious decisions because GPU workloads often have memory access rates that are an order of magnitude higher than those of CPU workloads. In particular, when both CPU and GPU workloads are identified as cache sensitive, the memory accesses from the GPU will pollute the shared cache, and wipe out cache blocks needed by the CPU. In such cases, it is desirable to give cache sensitive CPU workloads higher priority over cache sensitive GPU workloads to improve the overall performance.

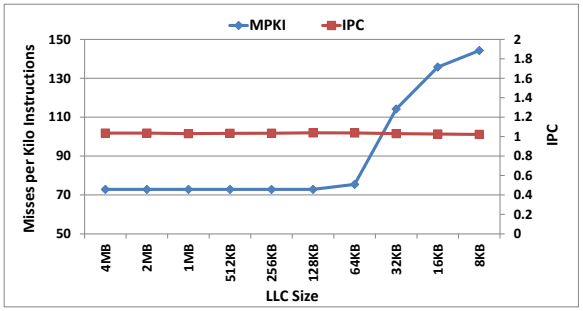



Fig. 3: Cache sensitivity of GPU application BoxFilter [Advanced Micro Devices Incorporated 2011]. BoxFilter exhibits a memory access behavior that is divergent from the traditional understanding of the cache sensitivity of streaming GPU applications.

### 2.2. Improving LLC sharing

In this work, we aim to address the challenges faced by existing cache management techniques in a heterogeneous multicore environment. First, we propose to use thread-level parallelism (TLP) as a runtime metric to correctly identify the cache sensitivity of GPU applications. Second, we propose to use LLC bypassing to improve cache management in the heterogeneous environment.

*2.2.1. Available TLP as a Runtime Metric.* The general cache insensitivity of GPU applications stems from two main reasons: i) streaming memory access behavior; and/or ii) high levels of available TLP. Even when

the memory access behavior is not streaming, GPU applications are able to tolerate higher memory access latency by utilizing the available TLP. Figure 4 shows the cache sensitivity and performance characteristics of the GPU application Floyd [Advanced Micro Devices Incorporated 2011]. Even when the application experiences increase in the cache miss rate, it is able to sustain performance to some extent due to the presence of enough threads that can prevent the GPU core from stalling for lack of data. Here, the average TLP at runtime is measured as the number of *wavefronts*[1] ready to be scheduled at any given time. Higher number of ready wavefronts indicate higher TLP, which in turn indicates that GPU can tolerate higher memory access latency.

These characteristics point to the fact that the TLP available in a GPU application is a good indicator to its cache sensitivity, and hence could aid in promoting an effective sharing of LLC among cores. Moreover, TLP is a true runtime metric that adapts to dynamic behaviors of the GPU application. To the best of our knowledge, no other work has directly utilized TLP as a metric to manage shared LLC in heterogeneous multicore processors. We observe that while mechanisms such as *set dueling* are not able to identify the true cache sensitivity of GPU applications, TLP forms an accurate metric for the same.

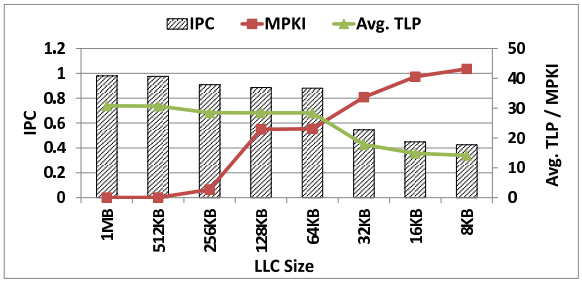



Fig. 4: TLP availability and cache sensitivity characteristics of GPU application Floyd [Advanced Micro Devices Incorporated 2011]. Figure shows that available TLP helps Floyd sustain its performance to some extent in the face of increasing MPKI.

*2.2.2. LLC Bypassing.* To improve the flexibility of cache management mechanisms in a heterogeneous multicore processor, we explore LLC bypassing techniques. Figure 5 shows the impact of bypassing[2] the shared LLC for 100%, 75%, 50%, and 25% of GPU memory access requests. The impact of LLC bypassing is related to the cache sensitivity of the application and the amount of TLP available at runtime. While applications such as Floyd, Gaussian, Mattran, which have low TLP, suffer from random bypassing, Bitsort and Hist remain unaffected due to their cache insensitivity. Applications like Dwthaar and Sobel, on the other hand, sustain their performance due to high TLP availability. On average, GPU applications can sustain up to 50% of LLC access bypassing without significant performance degradation. The GPU is able to do so by utilizing its high degree of available TLP.

LLC bypassing allows potentially different decisions for each incoming GPU access. When both CPU and GPU applications are identified as cache sensitive, the mechanism can consider various application characteristics while making the bypass decisions. Such characteristics include the difference in cache sensitivities of CPU and GPU applications, difference in memory access rate, and the amount of TLP available in the GPU application. Such fine-grained throttling of each LLC access can bring significant performance improvement as a result of better LLC utilization.

## 3. HETEROGENEOUS LLC MANAGEMENT

In this section, we describe our heterogeneous LLC management mechanism that mitigates the performance impact of LLC sharing by throttling LLC accesses initiated by the GPU cores. HeLM exploits the memory access latency tolerance capability of the GPU cores and allows the GPU cores to yield LLC space to the cache sensitive CPU cores without significantly degrading their own performance. In HeLM, we manage the

[1]Work is allocated to the GPU cores as *kernels* that contain a large number of threads. A kernel is further partitioned and mapped to different GPU cores as *thread-blocks* or *workgroups*. Scalar threads within each GPU core are scheduled simultaneously as *warps* [NVIDIA Corporation 2007] or *wavefronts* [Advanced Micro Devices Incorporated 2007] onto the SIMD computing engine.
[2]For 75%, 50%, and 25% bypassing, we randomly choose the GPU accesses for bypassing. The values shown are average for GPU benchmarks in Table III, executing on 4 GPU cores with specifications as mentioned in Table II.

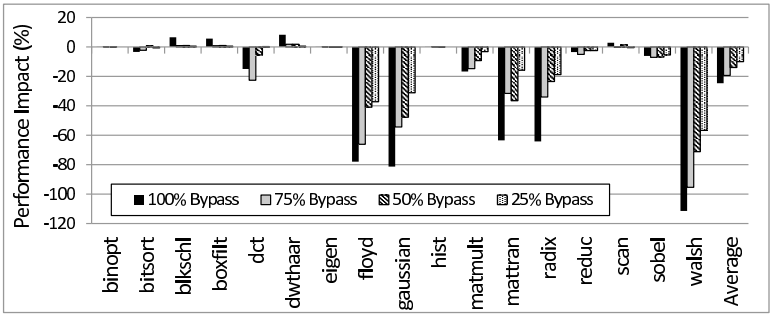



Fig. 5: Performance impact of bypassing LLC for memory accesses of GPU applications in Table III. Performance is relative to the performance of the application, without LLC bypassing, under LRU policy.

LLC occupancy of the GPU cores by allowing its memory traffic to selectively bypass the LLC, as shown in Figure 6, when: i) the GPU cores exhibit sufficient TLP to tolerate memory access latency; or ii) when the GPU application is not sensitive to LLC performance.

For each GPU memory access, the decisions for bypassing the LLC is made at the shared LLC. On an L1 cache miss, the TLP information of the GPU core is attached to the LLC access request. If the access misses at the LLC, the current TLP is compared to a selected threshold. If the current TLP is greater than the threshold, response to the cache miss bypasses LLC. The available TLP at runtime is measured using hardware performance monitors that measure the number of *wavefronts* ready to be scheduled at any given time. A higher number of ready wavefronts indicates higher TLP, which in turn suggests that the GPU can tolerate higher memory access latency.

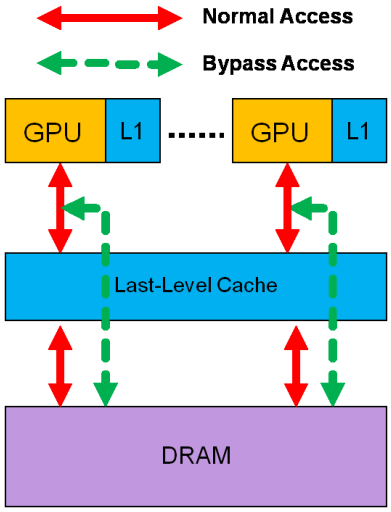



Fig. 6: LLC bypassing technique employed by GPU cores in HeLM.

Figure 7 shows the high level view of HeLM. GPU LLC bypassing decisions are made on the basis of the cache sensitivities of both CPU and GPU applications. The CPU application is given higher priority in our algorithm as it is, in general, more cache sensitive. If the CPU application is found cache sensitive, GPU memory accesses are subject to aggressive LLC bypassing. If not, GPU LLC sensitivity is considered and a bypass aggressiveness is selected accordingly. When neither of the applications are cache sensitive, the bypassing aggressiveness selected does not impact the performance. However, it could have significant impact on the energy consumption and bandwidth utilization, both on-chip as well as off-chip.

The cache sensitivity of the CPU and GPU applications plays a critical role in making bypass decisions. A cache-insensitive CPU application does not benefit from increased LLC space made available by GPU LLC bypassing. Bypassing LLC for a cache-sensitive GPU application executing along with such cache-insensitive CPU applications could degrade GPU performance without improving the overall performance.

In the following subsections, we discuss in detail the techniques employed to identify: i) the cache sensitivities of the CPU and GPU applications; and ii) an effective TLP threshold to measure the memory access latency tolerance of the GPU application. We combine these metrics into a *Threshold Selection Algorithm* (TSA) that makes GPU LLC bypass decisions.

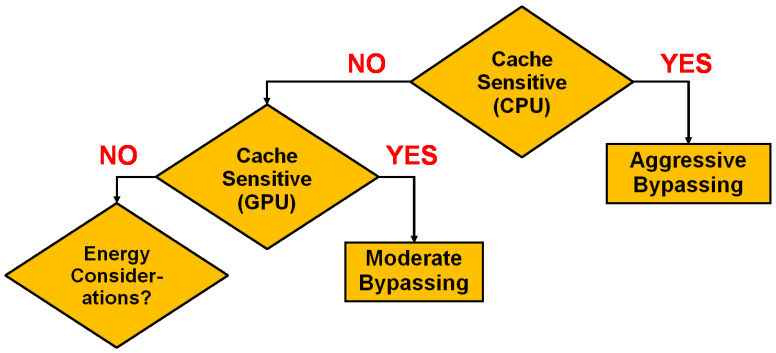



Fig. 7: Flowchart for bypass algorithm in HeLM.

### 3.1. Measuring Cache Sensitivity

We employ a mechanism based on the *set dueling* [Qureshi et al. 2007] technique to measure the cache sensitivity of the CPU and GPU applications. Set dueling applies two opposing techniques to two distinct sets, and identifies the characteristic of the application from the performance difference among the sets. Dynamic Set Sampling (DSS) [Qureshi et al. 2006] has shown that sampling a small number of sets in the LLC can indicate the cache access behavior with high accuracy. We use this technique by sampling 32 sets (out of 4096) to measure the cache sensitivity.

Figure 8 shows a high-level view of the workings of the set-dueling technique. Here, for a 4096 set cache structure, every 128$^{th}$ set starting from Set$_0$ follows POLICY1, while every 128$^{th}$ set starting from Set$_1$ follows POLICY2. Events such as cache misses are monitored, and these events increment a saturation counter when it occurs in POLICY1 sets and decrement the saturation counter when it occurs in POLICY2 sets. The value of the saturation counter is used to determine which policy is performing better. This policy is then applied on the *follower* sets.

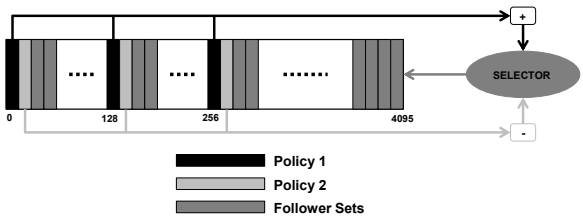



Fig. 8: Overview of the set-dueling [Qureshi et al. 2007] technique.

For HeLM, the two opposing policies used in set-dueling are: bypassing with *high aggressiveness* and bypassing with *low aggressiveness*. In HeLM, the bypassing aggressiveness is adapted by choosing an appropriate threshold for bypassing. The threshold relates to the available TLP in the GPU application. When the available TLP is higher than a selected threshold, the GPU application has enough parallelism to sustain increased memory access latency and is suitable for LLC bypassing.

For a GPU that supports up to 64 simultaneous wavefronts (warps), the threshold range is 0 to 63. In this range, we select two arbitrary thresholds: HighThr (say 47) and LowThr (say 15). HighThr corresponds to less aggressive bypassing as current TLP has to be higher than HighThr to enable bypassing. Similarly, LowThr corresponds to more aggressive bypassing.

*3.1.1. CPU LLC Sensitivity.* We evaluate the cache sensitivity of the CPU application by monitoring the impact of GPU LLC bypassing on the performance of the CPU application. Since CPU applications are more cache sensitive than GPU applications, change in cache miss rate typically directly affects the performance of CPU applications. We measure two CPU LLC misses *MissLow* and *MissHigh* corresponding to GPU bypassing at LowThr and HighThr respectively. Two *set dueling monitors* (SDM) are used at the LLC to obtain the MissLow and MissHigh numbers, each bypassing GPU accesses at LowThr and HighThr respectively. Since the GPU takes more LLC space with HighThr than with LowThr, MissHigh is always greater

than MissLow. If the difference between MissHigh and MissLow ($\Delta$MISS$_{CPU}$) is greater than *mThreshold*[3], GPU bypassing is affecting the CPU LLC behavior, and hence its performance. This criterion can also identify compute intensive as well as streaming CPU workloads[4].

*3.1.2. GPU LLC Sensitivity.* Figure 3 shows that cache miss rate is not a direct indicator of performance for GPU applications. Hence, we adapt the set-dueling technique to enable measuring GPU LLC sensitivity by directly measuring the performance of the GPU core. For this purpose, we utilize two GPU *sampling cores* and the two TLP thresholds: *LowThr* and *HighThr*. In every sampling period, one of the GPU cores (*LowGPU*) performs LLC bypassing at LowThr, while the other core (*HighGPU*) uses HighThr. LowThr is always smaller than HighThr and indicates a higher rate of bypassing. Hence, LowGPU bypasses more memory accesses than HighGPU. A significant performance difference ($\Delta$IPC$_{GPU}$), greater than *pThreshold*[5], between these two cores indicates that LLC bypassing is having an adverse impact on the GPU performance and hence the GPU application is cache sensitive. If the performance difference is within the limit, the GPU application is considered cache insensitive. This is similar to the *Core Sampling* technique used by TAP [Lee and Kim 2012], where sampling cores insert at MRU/LRU positions in the LLC. However, in the case of TAP, this leads to an unwanted side-effect as discussed in Section 5.3.3.

### 3.2. Determining Effective TLP Threshold

Determining the effective TLP threshold to initiate GPU LLC bypass is critical. To adapt to the diversity among GPU applications and the runtime variations within an application itself, we propose an algorithm to dynamically adapt LowThr and HighThr. Our heuristic is inspired by the *binary-chop* algorithm that is commonly used for searching an element in a sorted list bound by limits *MaxLimit* and *MinLimit*. Binary-chop algorithm starts with two parameters U and L such that U $\geq$ L, and calculates a decision element E as the average of U and L (AVG(U, L)). At the beginning of the algorithm, U and L are initialized to MaxLimit and MinLimit, respectively, and a prediction is made. If the decision element is lower than expected, the search window is moved up (GO UP) by updating U and L as shown in Table I. If the prediction is higher than expected, the search window is moved down (GO DOWN). At each step, E is recalculated, and the process is continued until E matches with the searched element.

| Action | U | L |
|---|---|---|
| INIT | MaxLimit | MinLimit |
| GO UP | AVG(MaxLimit, U) | E |
| GO DOWN | E | AVG(L, MinLimit) |

Table I: Binary-chop algorithm for adapting sampling thresholds at runtime.

Our adaptation of the binary-chop algorithm recomputes the higher and lower bypass thresholds at runtime depending upon the application behavior. We start by initializing HighThr (U) = $\frac{3}{4}\times$MAX$_{wavefronts}$, and LowThr (L) = $\frac{1}{4}\times$MAX$_{wavefronts}$. Here, MaxLimit = MAX$_{wavefronts}$, MinLimit = MIN$_{wavefronts}$. After every sampling period, HighThr and LowThr values are updated with new values of U and L, respectively.

Figure 9 demonstrates the working of this algorithm with an example. Here, we start with HighThr and LowThr as shown in PHASE 0. In each phase, the selected threshold is highlighted. If HighThr is selected and the program behavior indicates the need to reduce bypassing aggressiveness, binary-chop algorithm will perform the GO UP step, increasing the thresholds as shown in PHASE 1. If LowThr is selected and the program behavior indicates the need to increase bypassing aggressiveness, GO DOWN step will be taken. However, if there is a switch in direction from GO UP to GO DOWN, as shown in PHASE 2, we detect

[3]Based on empirical analysis, we set mThreshold to 10%.
[4]For compute intensive workloads, MissHigh $\approx$ MissLow $\approx$ 0, while for streaming workloads, MissHigh $\approx$ MissLow $\approx$ K (positive number) as shown in Figure 10.
[5]Based on empirical analysis, we set pThreshold to 5%.

a toggle and retain the previously selected threshold. A toggle indicates reaching of a stable phase and, as observed in PHASE 3, the thresholds are maintained for a specified number of sampling periods. After this, the algorithm restarts as shown in PHASE 4.

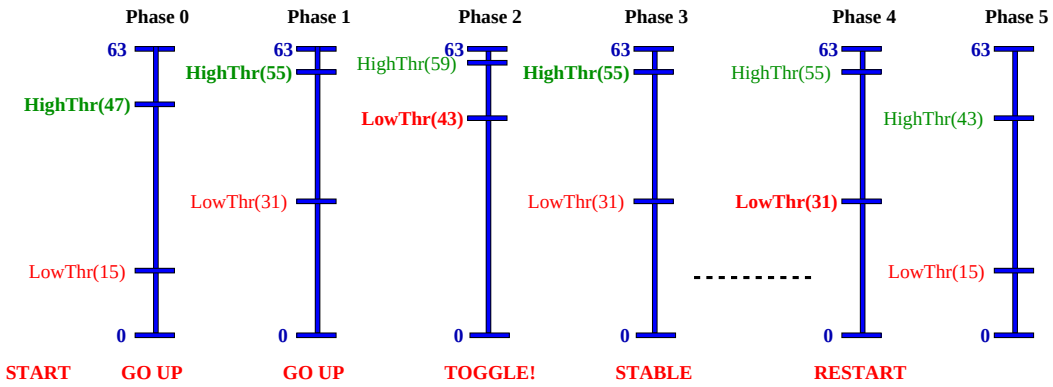



Fig. 9: Binary-chop algorithm for varying bypassing threshold.

### 3.3. Putting It All Together

We combine the cache sensitivity information and TLP thresholds into a *Threshold Selection Algorithm* (TSA). TSA monitors the workload characteristics continuously and re-evaluates the TLP threshold at the end of every sampling period (1M execution cycles in our study). Once the threshold is chosen, it is enforced on all the *follower* GPU cores. The pseudocode for TSA is shown in Algorithm 1. If the CPU application is cache sensitive, LowThr is selected for aggressively bypassing LLC for GPU memory accesses. Otherwise, the choice of threshold depends on the characteristics of the GPU application. If GPU application is cache sensitive, HighThr is selected for bypassing LLC for GPU memory accesses.

Based on the threshold selected by TSA, HighThr and LowThr are re-calculated using the binary-chop heuristic discussed in Section 3.2. For cache sensitive GPU applications, LLC bypassing aggressiveness is reduced by action GO UP; otherwise, the aggressiveness is increased by action GO DOWN. If the actions toggle between GO UP and GO DOWN for consecutive sampling periods, we maintain the existing HighThr and LowThr values for next five sampling periods.

CPU cache management policies have employed thread awareness to avoid the domination of one application on the sharing policy. Mechanisms such as thread-aware DIP [Jaleel et al. 2008] and thread-aware DRRIP [Jaleel et al. 2010] (referred to as DRRIP here) utilize separate set of SDMs to isolate the influence of applications on each other. Similarly, HeLM is made thread aware by assigning individual MissLow and MissHigh counters to calculate $\Delta\text{MISS}_{CPU}$ for each thread. For thread awareness, TSA selects LowThr as the TLP threshold if any of the $\Delta\text{MISS}_{CPU}$ is $\geq$ mThreshold.

### 3.4. Other Design Considerations

*3.4.1. Impact on On-Chip Energy and Off-Chip Access.* Allowing memory accesses that are unlikely to be reused in the cache to bypass the LLC can reduce the dynamic energy consumption of LLC accesses. However, LLC bypassing could also increase the off-chip DRAM accesses. Due to the streaming nature of GPU applications, the blocks in the LLC are mostly dead, and the accesses go off-chip to fetch data blocks from DRAM. Thus we observe that LLC bypassing does not increase off-chip DRAM accesses significantly. We evaluate the impact of bypassing on LLC energy consumption and off-chip accesses in Section 6.

**ALGORITHM 1:** Pseudocode for the Threshold Selection Algorithm (TSA).

```
Data: ΔIPC_GPU,ΔMISS_CPU
Result: Bypass TLP Threshold
if ΔMISS_CPU ≥ mThreshold then
  | Set LowThr as TLP threshold;
end
else if ΔIPC_GPU > pThreshold then
  | Set HighThr as TLP threshold;
end
else
  | if delta(ΔIPC_GPU,pThreshold) ≥ delta(ΔMISS_CPU,mThreshold) then
  |   | Set LowThr as TLP threshold;
  | end
  | else
  |   | Energy considerations decide TLP threshold;
  | end
end
```

*3.4.2. Handling Coherence.* The contemporary GPU does not support coherent memory hierarchy. However, if coherence is supported in future GPUs, bypassing can also be easily supported. The additional support for maintaining coherence with GPU bypassing may or may not be required depending upon the inclusion property of GPU cache hierarchy. Inclusion ensures that cache blocks present in high level caches are also present in the LLC, while non-inclusion or exclusion relaxes such a constraint. Bypassing essentially turns the inclusive LLC into a non-inclusive cache. Therefore, the support necessary for maintaining coherence in non-inclusive LLC can also be used to support bypassing in inclusive LLC. Coherence in non-inclusive caches is maintained by employing mechanisms such as snoop filter [Salapura et al. 2008], which is essentially a replica of higher level cache tags at the LLC. Therefore, bypassing for non-inclusive LLC will not require any modifications for handling coherence, while support similar to snoop filter will be necessary for inclusive LLC.

While this work evaluates workloads where the CPU and GPU applications have disjoint address spaces, we expect the proposed technique to be equally effective when these applications share the same address space. When data is shared between the CPU and the GPU, the underlying cache coherence mechanism will ensure correctness of data accesses, and the proposed bypass mechanism can be deployed with a snoop filter as discussed above. In this work, we model a GPU cache hierarchy that is write-through; however, it will work equally well with a write-back cache.

## 4. EXPERIMENTAL METHODOLOGY

We evaluate HeLM on a cycle accurate simulator, Multi2Sim [Ubal et al. 2012], that simulates both CPU and GPU cores as depicted in Figure 2. The CPU cores are modelled on a 4-wide out-of-order x86 processor, while the GPU cores are based on AMD Evergreen [Advanced Micro Devices Incorporated 2009] architecture. We extend the memory subsystem in Multi2Sim to support shared LLC between CPU and GPU cores. Table II shows the parameters of the cores we simulate.

We evaluate 500 million instructions for each CPU benchmark and 150 million instructions[6] for each GPU benchmark. The 500 million representative interval for each CPU benchmark, with *ref* input, is obtained through SimPoint [Hamerly et al. 2005] analysis. As followed in previous works [Qureshi and Patt 2006; Jaleel et al. 2010; Lee and Kim 2012], early finishing benchmarks continue to execute until all the benchmarks execute the specified number of instructions. We utilize McPAT [Li et al. 2009] and GPUWattch [Leng et al. 2013] for studying on-chip energy consumption, and DRAMSim2 [Rosenfeld et al. 2011] to calculate off-chip DRAM energy consumption.

[6] An instruction executed by all threads in a wavefront is counted as one instruction.

| CPU | |
|---|---|
| Core | 1-4 cores, 2.6GHz, 4-wide out-of-order, 64-entry RoB |
| L1 Cache | 4-way, 32KB, 64B line, private I/D (2 cycles) |
| L2 Cache | 8-way, 256KB, 64B line, unified (8 cycles) |
| **GPU** | |
| Core | 4 cores, 1.3GHz, 8-wide SIMD, 16K register file, 64 wavefronts, round-robin scheduling |
| L1 Cache | 4-way, 8KB, 64B line, private I/D (2 cycles) |
| Shared Memory | 32KB, 256B block (2 cycles) |
| **Shared Components** | |
| LLC | 32-way, 2-8MB, 64B line, 4-tiles (20 cycles) |
| DRAM | 4GB, 4 controllers (200 cycles) |
| NoC | Mesh topology, 32B flit-size |

Table II: Configuration parameters for the heterogeneous evaluation infrastructure.

Selecting an appropriate baseline processor model is necessary so as not to skew the evaluations in favor of any policy. We model our baseline design on AMD Fusion APU [Brookwood 2010] (A4-5300) which contains two x86 CPU cores and 128 AMD Radeon GPU stream processors (grouped into 4 compute units). We term this, the 2C4G configuration where 'C' stands for CPU core and 'G' stands for GPU core (compute unit). The die area taken by four GPU cores matches the die area taken by one CPU core. This allows fair comparison between performances of an application executing on a CPU core and an application executing on the GPU cores.

An inappropriately chosen configuration could bias the study of the impact of GPU cores on the performance of the CPU application shown in Figures 1(a) and 1(b). This bias could occur when the GPU cores are significantly larger than the CPU core, and as a result, overwhelm the LLC occupancy and performance. To avoid any such bias that could result from the 2C4G configuration, we consider two more configurations on the either side of it: 1C4G and 4C4G. In 1C4G, one CPU core shares the on-chip resources with four GPU cores, and in 4C4G, four CPU cores share the on-chip resources with four GPU cores.

### 4.1. Benchmarks

The CPU benchmarks evaluated belong to the SPEC CPU2006 benchmark suite [Spradling 2007]. The GPU benchmarks evaluated are OpenCL programs from AMD APP (Application Parallel Programming) v2.5 software development kit [Advanced Micro Devices Incorporated 2011]. We classify these benchmarks into different categories, depending on their cache performance, as shown in Table III.

| Category | CPU Benchmarks | GPU Benchmarks |
|---|---|---|
| Cache sensitive | bzip2, gcc, mcf, perlbench, dealII, omnetpp, astar, soplex, povray, h264 | matrix-multiplication, matrix-transpose, gaussian, fast-walsh transform, floyd-warshal |
| Cache insensitive | **Streaming:** libquantum, bwaves, milc, zeusmp, cactusadm, gemsfdtd, lbm, leslie3d<br>**Compute intensive:** gobmk, hmmer, sgromacs, jeng, gamess, calculix, tonto, namd | **Streaming:** histogram, radix sort, blackscholes, reduction<br>**Performance insensitive:** sobel, dwthaar, scanarray, dct, box filter<br>**Compute intensive:** binomial option, eigen, bitonic sort |

Table III: Classification of CPU and GPU benchmarks.

We evaluate multiprogrammed workloads on heterogeneous processors with core configuration as shown in Table IV. For the three processor configurations, namely 1C4G, 2C4G, and 4C4G, we consider three workload combinations by the same name. Each of the CPU core executes a CPU benchmark while all the four GPU cores execute the same GPU benchmark. Thus, 1C4G contains one CPU and one GPU application, while 2C4G contains two CPU applications executing along with one GPU application. Equal number of CPU and GPU benchmarks are selected from cache sensitive and insensitive categories. Workloads in Table IV are formed by randomly selecting benchmarks from each category. A processor with one CPU core shares a 2MB LLC with the four GPU cores, while processors with two and four CPU cores share 4MB LLC and 8MB LLC, respectively, with the four GPU cores.

| Core Configuration | Workloads |
|---|---|
| 1CPU + 4GPU (1C4G) | 100 |
| 2CPU + 4GPU (2C4G) | 40 |
| 4CPU + 4GPU (4C4G) | 30 |

Table IV: Heterogeneous workloads evaluated.

*4.1.1. Workload Characteristics.* A rethinking of the optimal sharing of on-chip resources in a heterogeneous multicore processor necessitates a reevaluation of the characteristics of the target workload. We analyze the performance characteristics of general purpose GPU applications for varying LLC sizes. Based on instructions per cycle (IPC) and misses per kilo-instruction (MPKI) characteristics, we broadly classify these applications as either cache-sensitive or cache-insensitive. Cache-insensitive applications can further be classified into three types: compute-intensive, performance-insensitive, or streaming. The first type puts very little pressure on the shared LLC. The last two types have high LLC access rate, however, cache size has hardly any influence on their performance. This characteristic is either due to their streaming access behavior or their TLP availability.

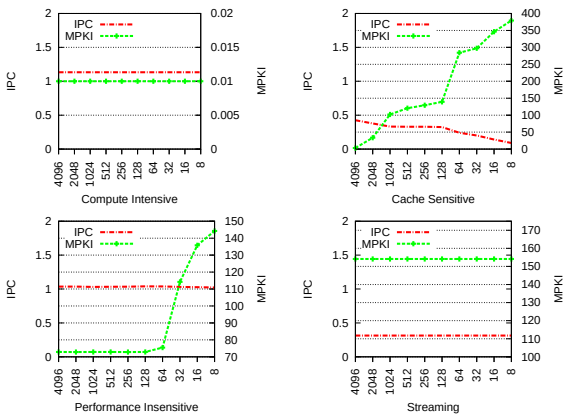



Fig. 10: LLC sensitivity of GPU benchmarks. Four different classes of GPU benchmarks are shown with their IPC and MPKI for different LLC sizes (in KB).

Figure 10 shows the IPC and MPKI characteristics for these categories. This characterization can be used to evaluate the effectiveness of LLC space on the application, and it shows that not all GPGPU applications have a uniform streaming access behavior. There are several cache-sensitive applications that need special attention while managing the LLC. These characteristics assume increased importance when GPU applications share the LLC with CPU applications in a heterogeneous multicore processor. Benchmarks belonging to each class are shown in Table III. We utilize this benchmark characterization to form the workload mix we evaluate. Section 5.5 evaluates HeLM based on these application classes.

### 4.2. Cache Management Policies

HeLM is decoupled from the underlying cache management policy, which brings flexibility to the mechanism as it can be adapted to work with any cache management policy. In this paper, we implement HeLM over DRRIP policy, however, it could well be implemented over other cache management policies such as Utility-based Cache Partitioning (UCP) [Qureshi and Patt 2006]. TAP was proposed with two variants: TAP-UCP and TAP-RRIP, built on top of UCP and DRRIP respectively. Since TAP-RRIP outperforms TAP-UCP in all evaluations, we choose to compare HeLM against TAP-RRIP. In DRRIP policy, incoming cache blocks are inserted at non-MRU position and are promoted later on cache hits. Similar to TAP, we do not promote

GPU blocks on LLC hits. Also, when both CPU and GPU blocks are available for replacement, we replace the GPU block first.

*4.2.1. Reuse-Based Cache Management.* Reuse-based mechanisms [Johnson et al. 1999; Lai et al. 2001; Liu et al. 2008; Kharbutli and Solihin 2008; Khan et al. 2010] have been studied extensively in CPU domain to improve cache utilization. Also known as dead-block predictors, they predict whether a cache block is dead or live and improve cache performance by replacing dead blocks first, by bypassing dead blocks, or by prefetching data into dead blocks. In this paper, we compare performance of HeLM with two reuse-based bypass mechanisms, MAT [Johnson et al. 1999] and Sampling Dead-Block Predictor (SDBP) [Khan et al. 2010].

MAT is an address-based reuse mechanism in which cache block reuse is determined using a *Memory Address Table* (MAT) at a *macroblock* granularity. MAT bypasses a cache block if the victim block has higher reuse than the one being inserted. SDBP, on the other hand, is a PC-based reuse mechanism in which the reuse pattern is learned from accesses to the cache blocks in a few sampled sets in the LLC. SDBP updates its prediction table using the PC of the last instruction that accesses a cache block in the sampled sets. On an access to the LLC, SDBP predicts the cache block as dead or live by referring to the prediction table, indexed by the PC of the instruction initiating the access. If the block being accessed is dead, it can be bypassed from the LLC. Other reuse-based mechanisms are discussed in Section 7.2.

## 5. EVALUATION

In this section, we evaluate the impact of HeLM on the performance of shared LLC in a heterogeneous multicore processor. We compare the performance of HeLM with DRRIP, MAT, SDBP, and TAP-RRIP, all normalized to LRU. MAT and SDBP were originally proposed for bypassing CPU memory accesses. However, to compare with HeLM, we employ these techniques to bypass only the GPU memory accesses.

### 5.1. Evaluation Metric

We use instructions per cycle (IPC) as the main performance metric. Speedup of each application (i) is calculated as shown in Eq. 1. Geometric mean (GM) of individual speedup (Eq. 2) gives the overall speedup for all the N applications in any configuration.

$$speedup_i = \frac{IPC_i}{IPC_i^{baseline}} \tag{1}$$

$$speedup = GM(speedup_{(1\text{ to }N)}) \tag{2}$$

### 5.2. Performance

We start our evaluation with the impact of these policies on the cache performance for CPU and GPU cores. Figure 11(a) shows the reduction in CPU and GPU LLC misses for these policies (normalized to LRU) for 1C4G, 2C4G, and 4C4G workloads. In case of CPU, although all of them perform better than LRU, the improvement for DRRIP and the reuse-based mechanisms (MAT and SDBP) are lower than TAP and HeLM. Additionally, HeLM outperforms TAP. Overall, HeLM reduces CPU LLC misses by 29.5%, 39.1%, and 33% over LRU for 1C4G, 2C4G, and 4C4G workloads, while the corresponding reduction for TAP are 13.9%, 18.1%, and 18.8%. On the other hand, TAP and HeLM increase the GPU LLC misses as they give lower priority to GPU over CPU. While DRRIP does better than TAP and HeLM, MAT and SDBP are the most favorable towards GPU.

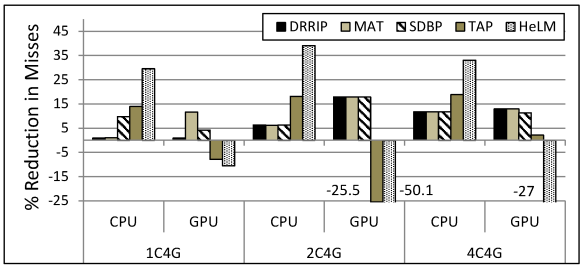
The performance impact of these policies on individual cores in each configuration is shown in Figure 11(b). For the CPU, it can be seen that the cache performance directly translates to speedup. Here, HeLM outperforms all other policies convincingly. The lower priority given to the GPU is evident in the speedup of TAP and HeLM. However, the impact of increased cache misses do not translate linearly into performance degradation for the GPU. This is due to the difference in cache sensitivity among the CPU and

GPU cores. This shows that it is preferable to prioritize CPU applications while managing the shared LLC space. In our experiments, the GPU benchmarks in 2C4G workloads show slightly higher overall performance degradation when compared to 1C4G and 4C4G workloads. This is due to the larger concentration of cache-sensitive GPU benchmarks in the 2C4G configuration, than 1C4G and 4C4G, on average. To avoid any bias towards HeLM and to maintain fairness in evaluations, benchmarks in each configuration are selected randomly. In our evaluations, 2C4G happens to have a larger proportion of cache-sensitive GPU benchmarks such as GAUSSIAN and FAST-WALSH TRANSFORM (Table III).

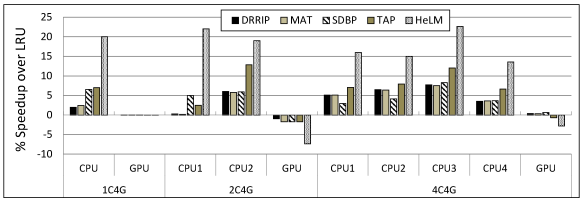
Combined speedup for CPU and GPU for all workloads is shown in Figure 11(c). Speedup is calculated as the geometric mean of individual benchmark speedups as mentioned in Eq. 2. The figure shows that HeLM outperforms all other replacement policies consistently in overall system performance. HeLM is also able to achieve performance improvement across configurations. Overall, HeLM performs 9.6%, 10.4%, and 12.5% better than LRU for 1C4G, 2C4G, and 4C4G workloads, respectively. The corresponding improvements in TAP over LRU are 3.4%, 4.5%, and 6.5%, respectively.

It can be noted that 2C4G and 4C4G configurations achieve higher overall speedup than 1C4G configuration, even when GPU performance is worse in these two compared to 1C4G (Figure 11(b)). This is expected as 2C4G and 4C4G workloads have more CPU applications that achieve higher performance gain and hence, by definition (Eq. 2), the overall performance is better in 2C4G and 4C4G configurations.

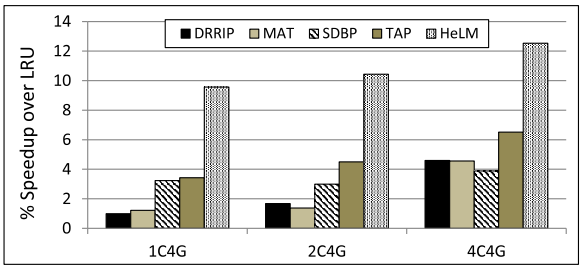
To form truly multiprogrammed workloads without any bias, we have randomly chosen applications, listed in Table III, for each workload. Although this method ensures fairness in performance analysis, we believe that it does not warrant comparison across different system configurations due to the differences in applications across workloads. However, performance of HeLM for each of these configurations shows that HeLM is able to achieve performance for varying number of cores in a heterogeneous multicore processor.




(a) Cache performance of CPU and GPU benchmarks.




(b) Speedup for CPU and GPU benchmarks.

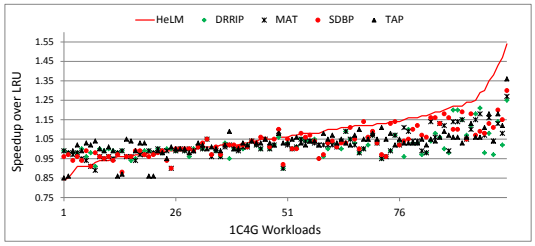



(c) Combined speedup for various workloads under different policies.

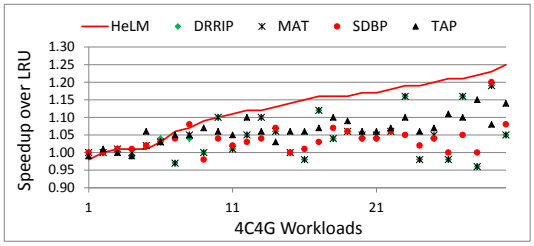
Fig. 11: Impact of various policies on cache performance and speedup of CPU and GPU benchmarks. Graphs show results for workloads: 1C4G, 2C4G, and 4C4G. Results are relative to the LRU policy.

*5.2.1. Detailed Performance Analysis.* Since each of the configurations consist of several workloads, the combined speedup across a configuration, as shown in Figure 11, presents limited information on the impact of various policies on individual workloads. In Figure 12, we present detailed information on a per-workload basis using s-curves[7]. The s-curves here present the performance of all the workloads in a configuration, for a policy, sorted by HeLM.

[7]For all s-curve figures, results are sorted by the performance of HeLM in ascending order.

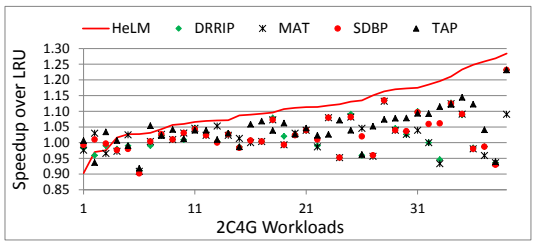



(a) S-curve for 1C4G speedup results.

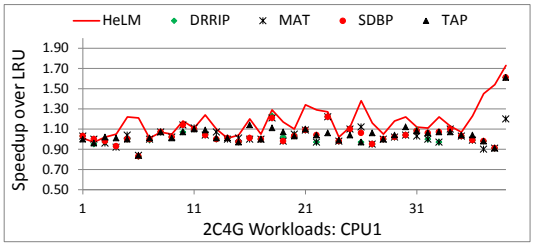



(b) S-curve for 4C4G speedup results.

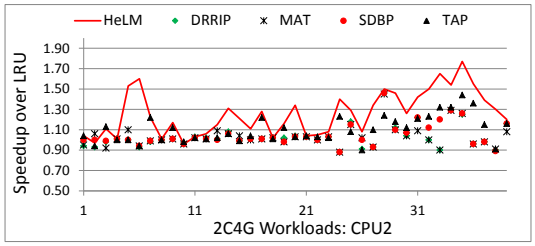
Fig. 12: S-curve for the combined speedup for various policies. Graphs show results for workloads: 1C4G and 4C4G. Results are relative to the LRU policy.



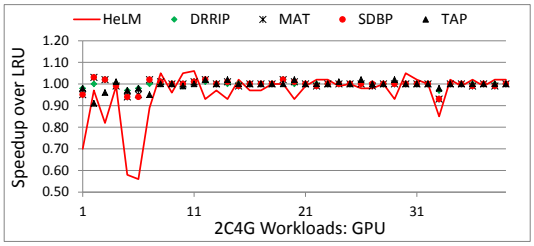

(a) S-curve for 2C4G combined speedup.




(b) S-curve for 2C4G CPU1.




(c) S-curve for 2C4G CPU2.




(d) S-curve for 2C4G GPU.

Fig. 13: S-curve of the combined speedup and speedup of individual cores in 2C4G under various policies. Results for individual cores in Figures 13(b), 13(c), and 13(d) are sorted by the combined speedup (13(a)) for direct comparison with overall performance. Results are relative to the LRU policy.

For 1C4G, there are several workloads, towards the left side of the s-curve (Figure 12(a)), where HeLM is outperformed by other policies. These are workloads where LLC bypassing in HeLM degrades the GPU performance significantly, while the performance benefit on the CPU-side is not substantial enough to improve the overall performance. As we scale towards 4C4G configuration (Figure 12(b)), we can observe that the number of workloads where HeLM is outperformed decreases. The larger number of CPU applications in this configuration increases the combined speedup.

To study the impact of various policies on individual applications in a workload, we study 2C4G configuration in detail (Figure 13). Here, separate graphs show the performance s-curve for individual applications, sorted by the overall IPC (Figure 13(a)) in ascending order. We can observe that the performance of HeLM for both the CPU applications is better than other policies. However, HeLM suffers performance degradation for certain GPU applications. A drop in overall performance in workloads on the left hand side of

s-curve, shown in Figure 13(a), can be directly correlated to a large drop in GPU performance, as shown in Figure 13(d).

### 5.3. Comparison with Other Policies

Here we discuss the reasons for the performance improvement of HeLM over various cache management policies we evaluate.

*5.3.1. DRRIP.* The effectiveness of DRRIP in multicore environment is visible in Figures 11(a) and 11(b) as DRRIP outperforms LRU. However, DRRIP faces difficulty in adapting to the heterogeneous characteristics of the cores. DRRIP policy, similar to LRU, does not consider the diversity among the on-chip cores and gives equal priority to both. Therefore, the higher LLC access rate from the GPU cores tends to skew the cache management policy in their favor. Thus, the performance improvement for the CPU core is limited, while GPU cores do not benefit much from the additional LLC space. Hence, the overall speedup for DRRIP in Figure 11(c) is low.

*5.3.2. MAT/SDBP.* The performance of reuse analysis based policies, MAT and SDBP, is very similar to DRRIP, however for different reasons. These mechanisms, although capable of LLC bypassing, are overly conservative in their approach towards the GPU applications. They detect reuse pattern in GPU memory access behavior and preserve the GPU blocks in LLC. This improves the cache performance of GPU as shown in Figure 11(a). However, GPU performance does not benefit significantly from the increased LLC space due to their TLP and the ability to tolerate higher memory access latency. This additional LLC space would have been better utilized had it been provided to the CPU application. Hence, these mechanisms also observe lower overall speedup than HeLM as shown in Figure 11(c).

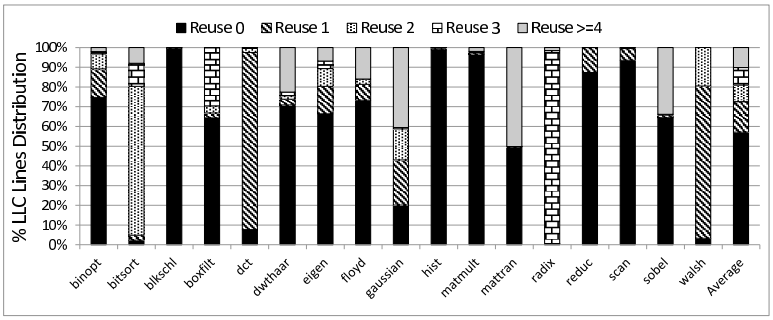



Fig. 14: Reuse characteristics of GPU applications at LLC. *Reuse N* indicates that a line in the LLC was accessed *N* times before it was evicted. In this study, GPU applications run alone and have access to a 2MB LLC.

Figure 14 shows the reuse characteristic of GPU applications. We can observe that these policies identify a significant amount of GPU cache lines, nearly 40% on average, to be reused. Correlating this with the bypass tolerance of GPU applications, as shown in Figure 5, it is clear that MAT and SDBP are overly conservative when it comes to bypassing GPU LLC accesses. For example, Boxfilt, Dwthaar, and Radix benchmarks are observed to have high reuse by MAT and SDBP, while they can sustain high levels of LLC bypassing. Reuse analysis could potentially be useful in improving LLC sharing in the presence of GPUs. However, our study suggests that we cannot rely on traditional reuse analysis techniques that were developed with CPU as the primary target. In SDBP, for example, PC-based mechanism is used for identifying reuse of the LLC lines. This mechanism is effective on CPU, where the applications have large number of instructions accessing the memory. However, SDBP tends to identify reuse in GPU more conservatively because of fewer number of memory instructions in the GPU kernels. This reduces the effectiveness of bypassing using SDBP in heterogeneous configurations.

*5.3.3. TAP.* TAP considers the diversity of on-chip cores in optimizing the cache management policy, and improves performance over existing policies by prioritizing CPU over GPU. However, HeLM still outperforms TAP for two reasons:
(i) the *core sampling* technique used by TAP leaves a significant portion of the shared LLC occupied by the GPU cores. Majority of these blocks originate from the GPU core that inserts at the MRU position as part of the core sampling technique. However, a significant portion of these blocks end up being dead blocks. In our experiments with 1C4G workloads, as shown in Figure 15, we observe that nearly 25% of the shared LLC space is occupied by the GPU dead blocks that were inserted at the MRU position. This leads to eviction of useful CPU blocks, leaving significant room for improvement. Since HeLM does not suffer from this side effect, it performs better than TAP.
(ii) TAP takes a binary decision on whether the GPU application is cache sensitive or not. This decision is then used to override the underlying policy for all the accesses in the sampling period. However, such a binary decision is at a coarse granularity, while a more fine-grained ability to control the LLC share between CPU and GPU could potentially improve the performance of both the cores. HeLM is able to control the cache occupancy of the GPU cores at a finer granularity, by taking bypass decision for each GPU access. This also helps in outperforming TAP.

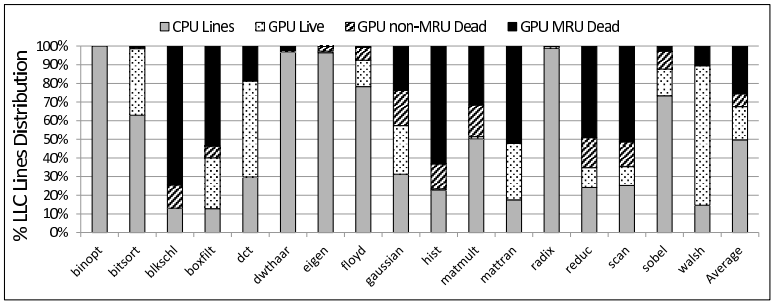



Fig. 15: Distribution of the shared LLC in TAP. A significant portion of the LLC is occupied by GPU deadblocks. This study is performed on a 1C4G configuration. The CPU core always executes 401.bzip2 application.

### 5.4. Sensitivity to Cache Size

Figure 16 presents the sensitivity of HeLM to varying LLC sizes. For this study, we chose the 4C4G configuration as it taxes the LLC the most. To configure different LLC sizes we vary the LLC associativity. As shown in the figure, HeLM outperforms other policies for all cache configurations. Although the performance benefits of HeLM is more evident with smaller LLC size, it is able to preserve its benefits with increasing LLC sizes. This shows that HeLM can adapt well to variations in cache configurations.

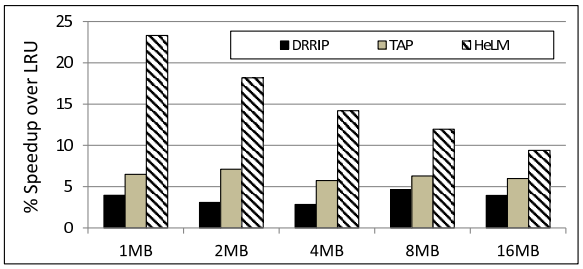



Fig. 16: Performance of 4C4G workloads with varying cache sizes. Result are relative to the LRU policy.

### 5.5. Workload Types

Mixing of CPU and GPU applications in a heterogeneous multicore processor creates workloads with unique characteristics. To evaluate the potential opportunities in these workloads, we broadly classify them based on their cache sensitivity, resulting in four different categories:

— *CPU cache Sensitive, GPU cache Insensitive* (CSGI): CSGI is perhaps the most common category in which GPU occupies majority of the LLC space leading to poor performance of the CPU application in existing cache policies.
— *CPU cache Sensitive, GPU cache Sensitive* (CSGS): Although GPU is cache sensitive in this combination, it has the advantage of high TLP. Hence, any additional LLC space given to CPU could bring a larger overall performance improvement.
— *CPU cache Insensitive, GPU cache Insensitive* (CIGI):
— *CPU cache Insensitive, GPU cache Sensitive* (CIGS): These categories do not leave significant room for performance improvement as the CPU applications are cache insensitive.

Figure 17 presents the speedup for the evaluated policies, over LRU, for the four categories. As expected, all the policies show performance improvement over LRU in the first two categories (CSGI, CSGS) where CPU is cache sensitive. Also, we can observe that HeLM outperforms all the other policies in these categories, particularly by a significant margin for CSGI which is the most common category. In the last two categories (CIGI, CIGS), there is hardly any performance improvement over LRU for any of the policies.

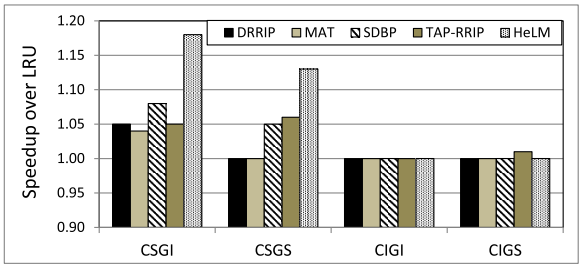



Fig. 17: Speedup for 1C4G workloads category-wise. Result are relative to the LRU policy.

### 5.6. Off-Chip Bandwidth Utilization

Allowing certain memory accesses to bypass the LLC can potentially increase the off-chip bandwidth utilization which could, in turn, impact the performance of the system. In our evaluations, we accurately measure the impact of this overhead using cycle accurate network and DRAM simulators as mentioned in Section 4. For 2C4G configuration, HeLM increases the off-chip bandwidth utilization by only 7% over LRU policy. This shows that HeLM is able to improve performance without significantly increasing the DRAM bandwidth utilization. It should be noted that TAP also increases off-chip bandwidth utilization by 3.4%.

In systems which utilize significant amount of DRAM memory, such as data center systems, the increase in off-chip memory bandwidth utilization due to bypassing could have a significant impact on performance. In such situations, HeLM should consider off-chip bandwidth utilization while making bypassing decisions.

| Policy | Hardware | Overhead |
|---|---|---|
| MAT [Johnson et al. 1999] | 4K-entry memory address table (each entry: 20-bit tag, 8-bit counter, 1 valid bit) | 14.5 KB |
| SDBP [Khan et al. 2010] | 4K-entry x 3 prediction tables (each entry: 2-bit counter), sampler sets | 13.7 KB |
| TAP [Lee and Kim 2012] | Instruction counters (20-bit x 4 GPU cores), core IDs (10-bit x 4 LLC tiles) | 120 bits |
| HeLM | MissHigh and MissLow counters (20-bit each), TLP threshold register (6-bit), instruction counters, core IDs | 166 bits |

Table V: Hardware Overhead.

### 5.7. Hardware Overhead

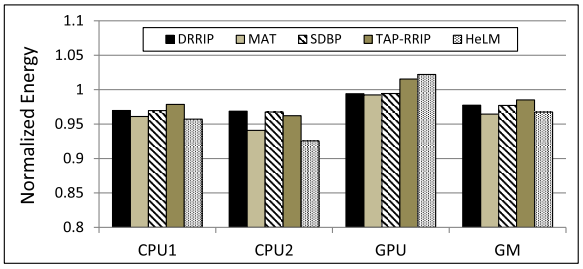
Table V presents the hardware overhead for various cache management policies, including HeLM. While MAT and SDBP require significant amount of storage to track the reuse of LLC blocks, TAP and HeLM can be implemented using simple hardware counters. HeLM utilizes MissHigh, and MissLow counters to track GPU and CPU LLC access behavior. Also, instruction counters and core IDs are required to identify the cache sensitivity of the GPU application. The TLP threshold register holds the TLP threshold selected by TSA. In summary, hardware overhead for HeLM is comparable to that for TAP, and both these mechanisms use significantly less additional hardware than MAT or SDBP.

## 6. ENERGY CONSUMPTION

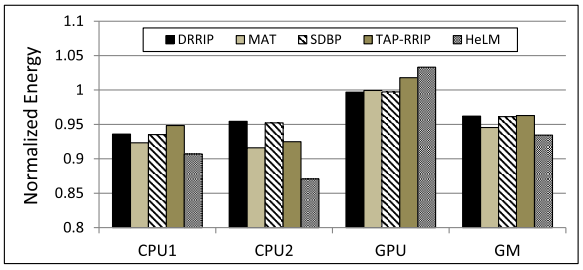
HeLM's LLC bypassing technique could alter the energy consumption profile of the system, both on-chip as well as off-chip. On-chip, there are two scenarios contributing to LLC dynamic energy: (i) on an LLC hit, tag array and cache block data are accessed; and (ii) on an LLC miss, tag array is accessed, a cache block is written back to memory if it is dirty, and data for the missed access is written to the cache block. When an LLC access is bypassed on a miss, only the tag array is accessed, eliminating the energy consumption of data block accesses. Additionally, dynamic access energy of the memory controller could also be altered by LLC bypassing because of the changes in off-chip access requests. Static energy, on the other hand, is dependent on the total execution time, and is hence related to the performance of the policy.

Here, we discuss the energy consumption of our baseline 2C4G configuration. The energy consumption shown in Figure 18 for each of the policy and configuration is normalized to the LRU policy. Figure 18(a) shows the on-chip dynamic energy consumption (cores, LLC, memory controller) for the various policies. HeLM improves the on-chip dynamic energy consumption by about 2% compared to TAP. Figure 18(b), on the other hand, shows the on-chip static energy consumption (cores, LLC, memory controller). Here also, HeLM improves the overall on-chip static energy consumption due to the performance improvement. Overall, Figure 18(c) shows the combined (static + dynamic) on-chip energy for the 2C4G configuration. HeLM consumes about 2% less on-chip energy than TAP.

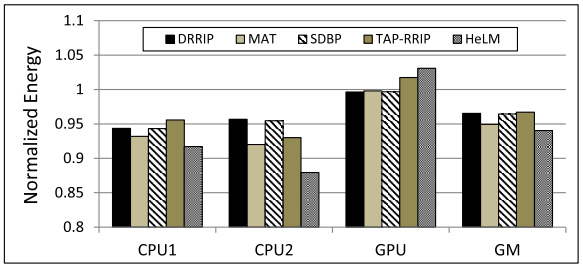
LLC bypassing, on the other hand, could lead to an increase in off-chip main memory




(a) On-chip dynamic energy consumption

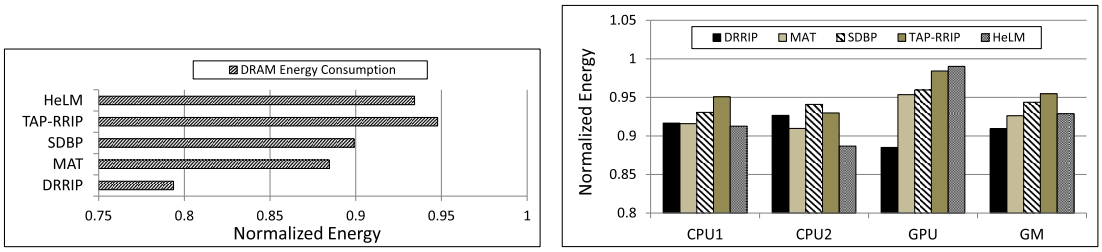



(b) On-chip static energy consumption




(c) On-chip total energy consumption

Fig. 18: On-Chip energy consumption for the 2C4G configuration. Figures show the energy consumption for two CPU applications, one GPU application, and for the system (geometric mean). Results are normalized to the LRU policy.




(a) DRAM energy consumption for the different policies for the 2C4G configuration. The results are normalized to the LRU policy.

(b) Total system (on-chip + DRAM) energy consumption for the different policies for the 2C4G configuration. The results are normalized to the LRU policy.

Fig. 19: Off-chip (DRAM) and total system energy consumption for the different policies for the 2C4G configuration.

(DRAM) accesses, resulting in a potential increase in DRAM dynamic energy consumption. Figure 19(a) shows the DRAM energy consumption for 2C4G workloads. HeLM increases the DRAM energy consumption when compared to DRRIP, MAT and SDBP. However, its DRAM energy consumption is comparable to that of TAP. TAP also increases DRAM energy consumption due to the increase in DRAM access rate as a result of the low priority given to GPU memory accesses. DRRIP policy shows a significant reduction in DRAM energy consumption as it prioritizes memory accesses from the GPU cores that reduces DRAM access.

Figure 19(b) shows the overall system (on-chip + DRAM) energy consumption for the 2C4G configuration. Here, we see that HeLM's energy consumption is similar to DRRIP and MAT, and in fact lower than TAP and SDBP. HeLM outperforms TAP in energy consumption by about 3%. This shows that energy consumption is not a concern for the bypassing-based HeLM.

### 6.1. Energy Delay-Squared Product

Energy efficiency has emerged as an important metric in architectural evaluations. Several works [Kumar et al. 2003; Brooks et al. 2000; Salapura et al. 2005] have turned to energy delay-squared product ($ED^2$) [Pénzes and Martin 2002] as the metric of choice to study energy efficiency as it considers both energy consumption and performance in determining the efficiency of a processor. In this section, we evaluate HeLM on the basis of the energy delay-squared product metric. For a multiprogrammed workload such as 2C4G, however, calculating $ED^2$ gets tricky. Below we describe how we calculate the $ED^2$ metric.

For our baseline 2C4G workload, $ED^2$ for each CPU application and the GPU application is calculated separately as they experience different execution times and performance improvements. Energy consumption for each core is calculated separately as they have different architectural characteristics. Meanwhile, energy consumed by shared resources, such as LLC and memory controllers, is pro-rated for each application on the basis of either execution time (for static energy) or access numbers (for dynamic energy). Geometric Mean (GM) is used to calculate the $ED^2$ for a policy for the entire workload. The $ED^2$ is normalized to the baseline of LRU policy.

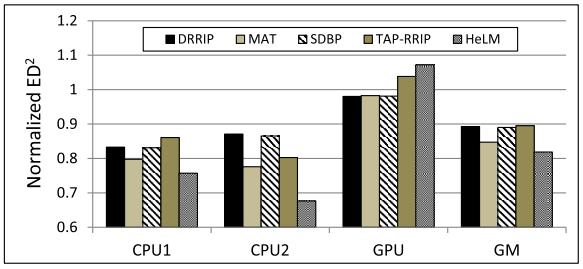



Fig. 20: Total system energy delay-squared ($ED^2$) product for the different policies for the 2C4G configuration. Results are normalized to the LRU policy.

Figure 20 shows the normalized $ED^2$ for all the policies for the 2C4G configuration. We see that for the CPU applications, HeLM's $ED^2$ value improves significantly compared to other policies, due to the performance improvement. For the GPU application, $ED^2$ worsens as we sacrifice GPU performance for overall performance improvement. However, we can note that HeLM's $ED^2$ for GPU is only slightly worse than that for TAP. Overall, HeLM performs better than other policies in terms of $ED^2$. It does nearly 8% better than TAP. These evaluations show that HeLM fares better than other policies from the energy efficiency perspective.

## 7. RELATED WORK

The importance of the cache subsystem to application performance has left a significantly large trove of research work in cache management techniques. In this section, we discuss only the works that are closely related to ours.

### 7.1. Cache Management

Existing works on cache management for homogeneous multicore processors can be divided into two general categories: (i) cache partitioning techniques; and (ii) cache replacement policies.

*7.1.1. Partitioning Techniques.* Stone et al. [Stone et al. 1992] conducted one of the first studies on optimal cache partitioning. However, they proposed static partitioning based on miss rate information for various applications with varying cache sizes. Dynamic cache partitioning mechanisms, on the other hand, aim to achieve their performance goal by dividing the cache ways among the applications at runtime. Suh et al. [Suh et al. 2004] introduce dynamic cache partitioning among threads executing on the same chip by utilizing hardware performance counters to maximize cache hit among threads. Moreto et al. [Moreto et al. 2008] propose a dynamic cache partitioning mechanism that considers the memory-level parallelism of an application and the impact of cache misses on its performance. Quality-of-service (QoS) considerations were addressed for multicore cache partitioning by Chang et al. [Chang and Sohi 2007], while fairness was considered by Kim et al. [Kim et al. 2004]. Utility-based cache partitioning (UCP) [Qureshi and Patt 2006] tries to find the optimal cache partitioning by prioritizing applications on the basis of benefit from cache over the demand for cache. Thrasher caging [Xie and Loh 2010] re-evaluates cache partitioning mechanisms in the presence of one or more thrashing applications. Recently, PriSM [Manikantan et al. 2012] has introduced a probabilistic shared cache management framework that considers various aspects such as cache hit ratio, fairness, and QoS.

*7.1.2. Replacement Policies.* Cache replacement policies aim to identify the appropriate position to insert a new cache block and to identify the right victim for replacement to achieve their performance goal. Qureshi et al. propose the dynamic insertion policy (DIP) [Qureshi et al. 2007] that overcomes the impact of thrashing behavior of certain applications on other applications in the workload. DIP achieves this by inserting cache blocks from thrashing workloads at LRU position to minimize their cache lifetime. Jaleel et al. [Jaleel et al. 2010] utilize re-reference interval prediction (RRIP) to develop a replacement policy that is both thrashing and scan resistant. PIPP [Xie and Loh 2009] is a cache management technique that combines insertion and promotion policies to utilize the benefits of cache partitioning and adaptive insertion. Both RRIP and PIPP show that inserting cache blocks at MRU position (near-immediate re-reference) is not optimal, while insertion of cache blocks at non-MRU position and promoting them on cache hits improves the cache utilization significantly. Pseudo-LIFO [Chaudhuri 2009] proposes a new family of cache replacement policies that is based on fill stack as opposed to the recency stack followed by previous policies. Recently, hierarchy awareness about other levels of cache has been introduced by CHAR [Chaudhuri et al. 2012] to improve the replacement policy applied at the LLC.

However, these mechanisms face significant challenge in the presence of diverse cores sharing the LLC. Hence, they cannot be directly adopted to heterogeneous multicore processors. A recent work, TAP [Lee and Kim 2012], adapts UCP and RRIP for heterogeneous multicore processors. However, HeLM outperforms these policies as discussed in Section 5.3.

### 7.2. Dead Block Predictors

Reuse-based cache management, often referred to as dead-block predictors, have been proposed in prior works [Johnson et al. 1999; Lai et al. 2001; Liu et al. 2008; Kharbutli and Solihin 2008; Khan et al. 2010]. Lai et al. proposed a dead-block predictor to prefetch data into L1 data cache [Lai et al. 2001]. While Kharbutli et al. proposed counting-based dead-block predictors [Kharbutli and Solihin 2008] that consider the number of accesses to a cache block, Cache Bursts [Liu et al. 2008] observes references to a cache block at MRU position to make dead block prediction. All prior works have only addressed the dead-block prediction issue for CPU workloads. These mechanisms cannot be directly adopted for heterogeneous workloads as they prove to be overly conservative for the GPU as discussed in Section 5.3.2.

## 8. CONCLUSIONS

The growing importance of data-parallel accelerator cores, such as GPU, has lead to their integration with CPU cores on the same die. Such architectures with heterogeneous processing cores present a significant challenge to optimal sharing of on-chip resources such as the LLC. Our heterogeneous LLC management mechanism, HeLM, monitors the TLP available in the GPU application and uses this information to throttle the GPU LLC access when the application has enough TLP to sustain longer memory access latency. This in turn provides an increased share of the LLC to the CPU application, thus improving its performance. HeLM monitors the cache sensitivity of both CPU and GPU applications in heterogeneous workloads, and achieves LLC sharing that improves overall system performance and energy efficiency.

We evaluate HeLM against: (i) existing shared LLC management techniques (LRU, DRRIP); (ii) reuse-based bypassing mechanisms (MAT, SDBP); and (iii) the only technique proposed for heterogeneous multicore (TAP). Evaluations are based on a heterogeneous multicore processor modelled after AMD APU processors currently available in market. HeLM outperforms all these mechanisms in overall system performance. HeLM improves over LRU policy by 10.4% and outperforms TAP by 5.9% for the baseline processor configuration with two CPU and four GPU cores. Evaluations across configurations show that HeLM is able to maintain performance under varying processor core mix. HeLM also outperforms competing policies in energy efficiency. HeLM consumes nearly 3% less total energy compared to TAP, while reducing $ED^2$ by 8%, for the baseline configuration. To summarize, HeLM outperforms other policies consistently in terms of both performance and energy efficiency. Additionally, HeLM achieves these characteristics without significant increase in off-chip bandwidth utilization. For the baseline configuration, HeLM increases the off-chip bandwidth utilization by only 7% over LRU policy.

Through this work, we demonstrate that judicious allocation of space among diverse cores is key to efficient sharing of on-chip LLC in a heterogeneous multicore processor. By effectively managing LLC distribution, with cache sensitivity awareness, HeLM is able to improve both performance and energy efficiency over currently proposed techniques.

### Acknowledgment

This work is supported in part by National Science Foundation grants CCF-0916583 and CPS-0931931. We would also like to thank Ragavendra Natarajan, Jieming Yin, and Carl Sturtivant for their suggestions to improve the paper.

## REFERENCES

Advanced Micro Devices Incorporated. 2007. ATI Stream Computing Programming Guide. (2007). http://www.amd.com.

Advanced Micro Devices Incorporated. 2009. Evergreen Family Instruction Set Architecture . (2009). http://www.amd.com.

Advanced Micro Devices Incorporated. 2011. AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK). (2011). http://developer.amd.com/sdks/amdappsdk/.

David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. 2000. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. *IEEE Micro* 20, 6 (Nov. 2000), 26–44. DOI:http://dx.doi.org/10.1109/40.888701

Nathan Brookwood. 2010. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. *Advanced Micro Devices(AMD) White Paper* (2010).

Jichuan Chang and Gurindar S. Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS '07)*. ACM, New York, NY, USA, 242–252. DOI:http://dx.doi.org/10.1145/1274971.1275005

Mainak Chaudhuri. 2009. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 401–412. DOI:http://dx.doi.org/10.1145/1669112.1669164

Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. 2012. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 293–304. DOI:http://dx.doi.org/10.1145/2370816.2370860

Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism*.

Intel Corporation. 2009. Intel Sandy Bridge Microarchitecture. (2009). http://www.intel.com.

Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 208–219. DOI:http://dx.doi.org/10.1145/1454115.1454145

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 60–71. DOI:http://dx.doi.org/10.1145/1815961.1815971

T.L. Johnson, D.A. Connors, M.C. Merten, and W.-M.W. Hwu. 1999. Run-time cache bypassing. *IEEE Trans. Comput.* 48, 12 (dec 1999), 1338 –1354. DOI:http://dx.doi.org/10.1109/12.817393

Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. 2010. Sampling Dead Block Prediction for Last-Level Caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 175–186. DOI:http://dx.doi.org/10.1109/MICRO.2010.24

Mazen Kharbutli and Yan Solihin. 2008. Counter-Based Cache Replacement and Bypassing Algorithms. *IEEE Trans. Comput.* 57, 4 (April 2008), 433–447. DOI:http://dx.doi.org/10.1109/TC.2007.70816

Khronos Group. 2009. OpenCL - The open standard for parallel programming of heterogeneous systems. (2009). http://www.khronos.org/opencl/.

Seongbeom Kim, Dhruba Chandra, and Yan Solihin. 2004. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, Washington, DC, USA, 111–122. DOI:http://dx.doi.org/10.1109/PACT.2004.15

R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. 2003. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. 81–92. DOI:http://dx.doi.org/10.1109/MICRO.2003.1253185

An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*. ACM, New York, NY, USA, 144–154. DOI:http://dx.doi.org/10.1145/379240.379259

Jaekyu Lee and Hyesoon Kim. 2012. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. DOI:http://dx.doi.org/10.1109/HPCA.2012.6168947

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 *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 487–498. DOI:http://dx.doi.org/10.1145/2485922.2485964

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 469–480. DOI:http://dx.doi.org/10.1145/1669112.1669172

Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 222–233. DOI:http://dx.doi.org/10.1109/MICRO.2008.4771793

R Manikantan, Kaushik Rajan, and R Govindarajan. 2012. Probabilistic shared cache management (PriSM). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 428–439. http://dl.acm.org/citation.cfm?id=2337159.2337208

Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. 2013. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 225–234. http://dl.acm.org/citation.cfm?id=2523721.2523753

Miquel Moreto, Francisco. J. Cazorla, Alex Ramirez, and Mateo Valero. 2008. MLP-Aware Dynamic Cache Partitioning. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*. Lecture Notes in Computer Science, Vol. 4917. 337–352.

NVIDIA Corporation. 2007. NVIDIA CUDA C Programming Guide. (2007). http://www.nvidia.com.

Paul I Pénzes and Alain J. Martin. 2002. Energy-delay Efficiency of VLSI Computations. In *Proceedings of the 12th ACM Great Lakes Symposium on VLSI (GLSVLSI '02)*. ACM, New York, NY, USA, 104–111. DOI:http://dx.doi.org/10.1145/505306.505330

Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 381–391. DOI:http://dx.doi.org/10.1145/1250662.1250709

Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, Washington, DC, USA, 167–178. DOI:http://dx.doi.org/10.1109/ISCA.2006.5

Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 423–432. DOI:http://dx.doi.org/10.1109/MICRO.2006.49

P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters* 10, 1 (2011), 16–19. DOI:http://dx.doi.org/10.1109/L-CA.2011.4

Valentina Salapura, Randy Bickford, Matthias Blumrich, Arthur A. Bright, Dong Chen, Paul Coteus, Alan Gara, Mark Giampapa, Michael Gschwind, Manish Gupta, Shawn Hall, Ruud A. Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V. Kopcsay, Martin Ohmacht, Rick A. Rand, Todd Takken, and Pavlos Vranas. 2005. Power and Performance Optimization at the System Level. In *Proceedings of the 2nd Conference on Computing Frontiers (CF '05)*. ACM, New York, NY, USA, 125–132. DOI:http://dx.doi.org/10.1145/1062261.1062262

Valentina Salapura, Matthias Blumrich, and Alan Gara. 2008. Design and implementation of the Blue Gene/P snoop filter. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE, 5–14.

Cloyce D. Spradling. 2007. SPEC CPU2006 Benchmark Tools. *SIGARCH Computer Architecture News* 35 (March 2007). Issue 1.

Harold S. Stone, John Turek, and Joel L. Wolf. 1992. Optimal partitioning of cache memory. *Computers, IEEE Transactions on* 41, 9 (1992), 1054–1068. DOI:http://dx.doi.org/10.1109/12.165388

G.E. Suh, L. Rudolph, and S. Devadas. 2004. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing* 28 (2004), 7–26. Issue 1. DOI:http://dx.doi.org/10.1023/B:SUPE.0000014800.27383.8f

Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*.

Yuejian Xie and Gabriel H. Loh. 2009. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*. ACM, New York, NY, USA, 174–183. DOI:http://dx.doi.org/10.1145/1555754.1555778

Yuejian Xie and Gabriel H. Loh. 2010. Scalable shared-cache management by containing thrashing workloads. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'10)*. Springer-Verlag, Berlin, Heidelberg, 262–276. DOI:http://dx.doi.org/10.1007/978-3-642-11515-8_20