# Adaptive Cache Partitioning on a Composite Core

Jiecao Yu, Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Scott Mahlke
Computer Engineering Lab
University of Michigan, Ann Arbor, MI
{jiecaoyu, lukefahr, shrupad, reetudas, mahlke}@umich.edu

## 1. INTRODUCTION

In modern processors, power consumption and heat dissipation are key challenges, especially for battery-limited mobile platforms. Heterogeneous multicore systems are an effective way to trade increased area for improved energy efficiency. These systems consist of multiple cores with differing performance and energy characteristics[1]. Applications are mapped to the most energy-efficient cores that meet the performance requirements.

For these systems, application migration at a finer granularity can help expose more opportunities to improve energy efficiency. However, traditional systems with private L1 caches must explicitly migrate the L1 cache state when switching, incurring high overhead and limiting the switching to a coarse granularity. To reduce this migration overhead, the Composite Core[2] is proposed. By sharing much of the architectural states between heterogeneous pipelines ($\mu$Engines) within a single core, it reduces the migration overhead to near zero, allowing a single thread to switch between the $\mu$Engines at a fine granularity (on the order of a thousand instructions). In addition, Composite Cores can be augmented to support multithreaded execution.

When multiple threads are running simultaneously on a multicore system, they will compete with each other for shared resources. For example, in the case of shared caches, both threads are competing for the cache capacity, which may degrade overall performance.

Previous works (e.g., Vantage[3], UCP[4], PriSM[5]) have extensively studied cache partitioning mechanisms on L2 or lower levels of caches to help resolve the cache contention issue. However, in a Composite Core or SMT cores, threads share even latency critical L1 caches. On mobile platforms, if a foreground process and a background process (regarded as the primary and secondary threads, respectively) are running on a Composite Core, the foreground process may suffer from a performance loss (e.g., more than 5%) due to the cache contention with the background process on the L1 level.

We find that both the memory characteristics of the application phases and the microarchitecture of the $\mu$Engine the application is running on impact cache allocation at the L1 level. Therefore, this paper introduces an adaptive cache management scheme that focuses on the shared L1 caches for a Composite Core. This scheme is designed to limit the performance impact on the primary thread from a secondary thread due to cache contention. It employs way-partitioning and a modified LRU policy that overcomes limitations exclusive to L1 caches. Since threads can switch between $\mu$Engines at a fine granularity, the scheme must also be enabled to dynamically resize cache capacities at that granularity to capture changing cache demands.

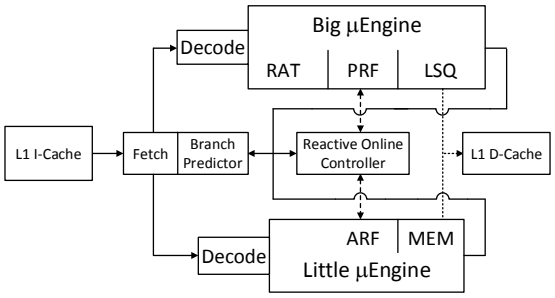


**Figure 1: Structure of Composite Core[2]**

## 2. BACKGROUND AND MOTIVATION

A Composite Core[2] is composed of an out-of-order pipeline (Big $\mu$Engine) and a in-order pipeline (Little $\mu$Engine) that share L1 caches as shown in Figure 1. The Big $\mu$Engine has higher performance but higher power consumption compared with the Little $\mu$Engine. The two $\mu$Engines share the front-end and the same L1 instruction and data caches, which dramatically reduces switching overheads.

In an augmented Composite Core supporting multithreaded execution, one thread is mapped to each $\mu$Engine. The controller exchanges threads between $\mu$Engines if it predicts this will yield improved energy efficiency gains.

With multiple threads running simultaneously on a Composite Core, cache partitioning is essential to maximizing system performance. Memory intensive threads can occupy most of the cache capacity, starving other threads through limited cache space. In the extreme case, threads thrash each other as they contend for overlapped cache space. Therefore, several cache partitioning mechanisms have been proposed to manage cache resources across processes.

There are two general methods to implement cache partitioning: controlling cache line placement (indexing mechanisms) or replacement (replacement-based mechanisms). Indexing mechanisms resize cache capacities by directly repositioning cache lines to the cache space assigned to corresponding threads, which incurs high migration overhead and makes smooth resizing difficult[6]. Here resizing means redeciding the percent of total cache capacity assigned to each thread. By contrast, replacement-based mechanisms resize cache capacities only by controlling the cache line replace-

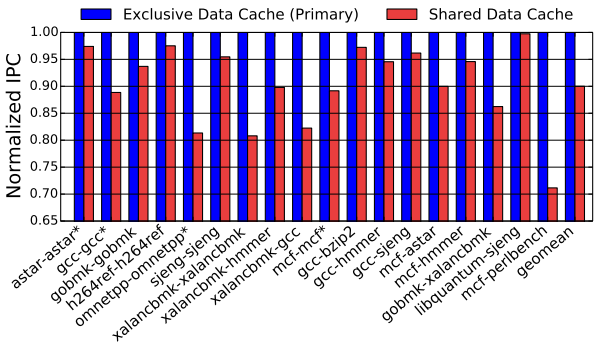


**Figure 2: Performance of the primary thread**

ment policy and therefore have lower resizing penalty.

Existing cache partitioning mechanisms focus on partitioning of L2 or last level caches (LLCs). However, for a Composite Core style architecture, the performances of both threads, especially the primary thread, can suffer from cache contention in L1 caches.

Figure 2 shows the performance loss of the primary thread caused by the L1 data cache contention. Every bar is labeled by the used benchmarks in which the former is the primary thread and the latter is the secondary thread, e.g., astar-astar*. Benchmarks marked with an asterisk use different inputs. The baseline is the performance of the primary thread when the L1 data cache can only be accessed by the primary thread. It shows that in more than half the cases, the primary thread has a performance loss of more than 5%. The performance loss can be 28% in the worst case and is 10% on average.

A naive solution to prevent the performance loss of the primary thread is assigning the L1 data cache exclusively to the primary thread as the baseline does in Figure 2. However, this strategy will dramatically decrease the total throughput and the performance of the secondary thread. Therefore, a cache partitioning scheme in the L1 caches becomes necessary to maximize the total throughput or the performance of the secondary thread while limiting the performance loss of the primary thread to be lower than 5%.

Ideally, cache partitioning can significantly reduce the cache contention that results from the overlap of working sets. However, the L1 cache size is usually smaller than the working set of either thread. Thus compared with working sets overlapping, the relations between memory sets accessed by threads in fine-grained instruction phases have more tight correlation with the cache contention penalty. Therefore, L1 cache partitioning mechanisms should focus more on fine-grained memory sets of different threads. Additionally, for a Composite Core, microarchitecture performance asymmetry further complicates the challenge of minimizing cache contention.

# 3. L1 CACHE PARTITIONING

To address these difficulties, we propose an adaptive way-partitioning scheme for the L1 cache which works at the fine granularity needed by thread switching on a Composite Core.

The main challenge specific to L1 caches is hit latency. As L1 caches interface directly with the processor, their hit latency must not exceed one (or possibly two) cycles. Therefore, it is difficult for the L1 caches to achieve an associativity higher than four. Complex way-hashing functions cannot be employed due to latency limitations. Besides, the latency limits the sizes of L1 caches to be smaller than application working sets for both instruction and data caches. Finally, most existing cache partitioning schemes do not provide a limitation on the performance loss of the primary thread. For these reasons, many existing cache partitioning algorithms are not well suited for L1 caches.

## 3.1 Heterogeneous Memory Accesses

As threads with different performance characteristics execute on different $\mu$Engines of a Composite Core, there is an inherent performance asymmetry.

For L1 caches, the inherent heterogeneity of Composite Cores should be taken into consideration. When running on the Big $\mu$Engine, a thread tends to generate more cache accesses due to increased issue width and advanced hardware structures. Contrarily, a thread running on the Little $\mu$Engine is constrained to issue memory access much slower. An L1 cache partitioning scheme must account for these heterogeneous memory access patterns, along with data-reuse rates and memory set sizes.

## 3.2 L1 Cache Way-Partitioning

Limitations on L1 caches require a cache partitioning mechanism with low accessing and resizing overheads. To smoothly resize the cache capacity, a way-partitioning algorithm based on an augmented cache replacement policy can be a suitable choice for L1 cache partitioning.

In our design, two threads are running simultaneously on the Composite Core. Each way of the L1 caches can be either assigned exclusively to thread 0, exclusively to thread 1, or shared by both threads. If all 4 ways are shared by both threads, the cache is identical to a traditional non-partitioning L1 Cache. In the situation that all ways are assigned to a single thread, the other thread must bypass the L1 caches to directly access data in the L2 cache.

Our way-partitioning scheme is based on an augmented Least Recently Used (LRU) policy. There is no data migrated when resizing the cache capacity for each thread. After resizing, all the cache blocks in the ways assigned to another thread are still kept in the same position. When the Composite Core accesses the cache, all 4 ways are searched to locate the data, similar to the non-partitioning cache. Therefore, the way-partitioning algorithm does not introduce extra overhead for hit latency. However, on a miss, when the cache must find a victim, the partitioning controller will only select a LRU block located in a way that is either assigned to that thread or shared by both threads. By doing this, it protects the data in the ways assigned to a specific thread from evictions by the other thread.

## 3.3 Adaptive Scheme

To capture changes of cache access behavior across instruction phases, our scheme can resize the cache capacity for each thread at a fine granularity (on the order of 10K to 100K instructions) during execution.

After one instruction phase finishes, the partitioning controller will re-evaluate which ways to assign exclusively to thread 0, thread 1, or to share between threads based on some characteristics of threads. These characteristics include cache reuse rates, memory set sizes and which $\mu$Engine

| Metrics | | Little | | | |
|---|---|---|---|---|---|
| | | HrHs | HrLs | LrHs | LrLs |
| Big | HrHs | (↑,↓) | (↑,↓) | (↑,↓) | (↑,↓) |
| | HrLs | (=,↑) | (=,=) | (=,↓) | (=,↓) |
| | LrHs | (↓,↑) | (↓,=) | (↓,↓) | (↑,↓) |
| | LrLs | (↓,↑) | (=,↑) | (=,↓) | (↓,↓) |

**Table 1: Cache Partitioning Priority based on cache access metrics (Hr/Lr: High/Low cache reuse rates; Hs/Ls: High/Low size of the memory set). The arrows in the tuples (e.g., (↑,↓)) represent the priorities of threads on the Big and Little $\mu$Engines, respectively. "↑": raise priority; "↓": lower priority; "=": maintain priority.**

the thread is running on. We use cache partitioning priority to describe the priority of one thread when resizing the cache capacities. Ways belonging to the thread with a low priority may become shared or be assigned to the other thread if it has a higher priority.

Table 1 shows the cache partitioning priorities varying with different characteristics. Note that in this paper, thread switching is disabled for simplicity and the primary thread will always execute on the Big $\mu$Engine. As shown in this table, threads that need more private cache resources tend to have higher priorities. For example, when the primary thread running on the Big $\mu$Engine has a high cache reuse rate and a small memory set, it is likely to be compute intensive. Therefore, when the secondary thread on the Little $\mu$Engine has a high cache reuse rate and a large memory set, the partitioning controller may decide to maintain the same cache ways for the primary thread and assign more cache ways to the secondary thread. However, if two threads both have a high cache reuse rate and a large memory set, the primary thread should get a higher priority due to the inherent heterogeneity of the Composite Core. Actually, the primary thread always tend to have a higher priority than the secondary thread, which can also help prevent the primary from a significant performance loss.

Besides the priorities of threads, the adaptive scheme must take the performance loss of the primary thread into consideration. It should limit the performance loss of the primary thread to at most 5%. Every resizing decision based on Table 1 is made under this performance loss limitation. Any cache partitioning modes that predicted to cause a significant performance loss in the primary thread will not be adopted.

# 4. RESULTS AND ANALYSIS

## 4.1 Experimental Methodology

To evaluate the adaptive cache partitioning scheme, the Gem5 Simulator[7] is extended for fine-grained resizing of cache capacities. Benchmarks studied are selected from the SPEC CPU2006 suite. Two benchmarks are combined to form a single multiprogrammed workload. Each workload is named as Benchmark1-Benchmark2 in which Benchmark1 is the primary benchmark and Benchmark2 is the secondary one. Benchmarks marked with an asterisk use different inputs. All benchmarks are compiled using gcc with -O2 optimizations for the ARM ISA. All workloads are evaluated with a fast forwarding for 1.5 billion cycles before beginning

| Architecture Features | Parameters |
|---|---|
| Big $\mu$Engine | 3 wide Out-of-Order @ 1.0 GHz<br>12 stage pipeline<br>92 ROB entries<br>144 entry register file<br>Tournament branch predictor (Shared) |
| Little $\mu$Engine | 2 wide In-Order @ 1.0 GHz<br>8 stage pipeline<br>32 entry register file<br>Tournament branch predictor (Shared) |
| Memory System | 32KB 4-way L1 I-Cache (Shared)<br>64KB 4-way L1 D-Cache (Partitioning)<br>1MB L2 Cache, 18 cycle access<br>4096MB Main Mem, 80 cycles access |

**Table 2: Experimental Hardware Parameters**

detailed simulations for 100 million instructions.

Table 2 gives more specific simulation configurations for the Composite Core and the memory system. The Big and Little $\mu$Engines are models as a 3-wide out-of-order pipeline and a 2-wide in-order pipeline, respectively. The adaptive cache partitioning scheme is implemented in the data cache while the instruction cache is directly shared between the two threads. Thread switching between the Big and the Little $\mu$Engine is disabled for simplicity.

Cache capacities are resized for every fix-length instruction phase. For different instruction phases, the cache partitioning scheme can switch between six different modes. In mode 0, all cache ways are shared between two threads. In mode 1, 2, 3 and 4, the partitioning controller assign one, two and three ways exclusive to the primary thread, respectively. Remaining cache ways, if any, are assigned exclusively to the secondary thread. In mode 5, two ways are shared by the two threads and two ways left are assigned exclusively to the primary and secondary threads, one per thread.

In this paper, the simulation is configured in an oracle way. For the oracle simulation, every instruction phase runs under all possible cache partitioning modes and the mode that maximizes the total throughput is chosen. Therefore, this oracle simulation actually finds the local optimal cache partitioning mode for every specific instruction phase but not the global optimal one for the entire simulation.

The length of the instruction phases is set to 100K instructions from the primary thread. Shorter instruction phase cannot capture the long-term impact of every cache partitioning mode and may lead to a decrease in total throughput. Therefore, taking history performance information into consideration when resizing the cache capacities can help learn more about the long-term impacts and may decrease the optimal length of the instruction phases.

To prevent the primary thread from losing much performance, the scheme keeps a performance limitation on the cache capacities resizing. The workloads are first executed with all the cache ways assigned to the primary thread. The performance of the primary thread in every workload is recorded as the baseline. In the simulation with adaptive cache partitioning scheme, the chosen cache partitioning mode for every instruction phase should not decrease the total throughput to be lower than 95% of the baseline

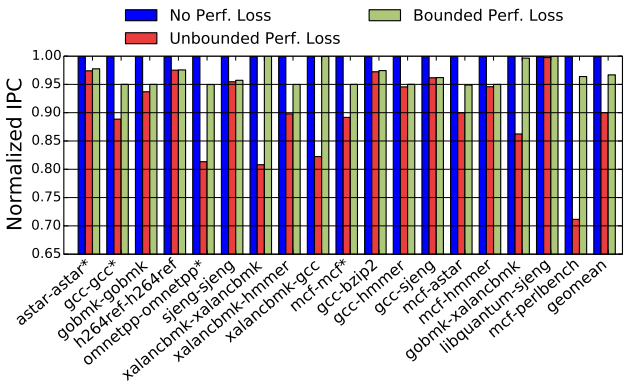


**Figure 3: Primary thread performance of No Performance Loss(NPL), Unbounded Performance Loss(UPL), Bounded Performance Loss(BPL)**

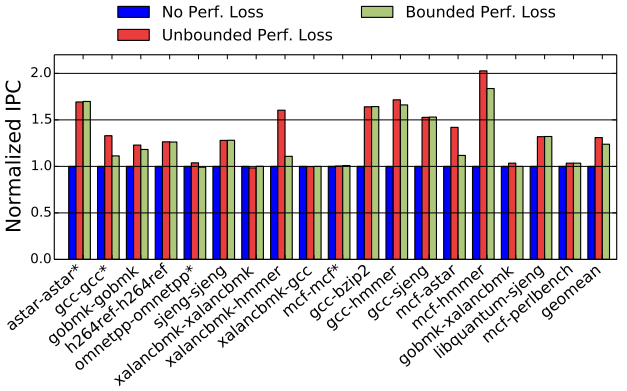


**Figure 4: Total throughput of NPL, UPL, BPL**

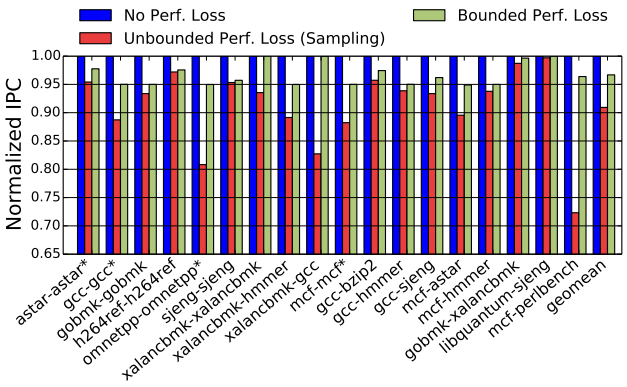


**Figure 5: Primary thread performance of NPL, UPL (Sampling), BPL (Adaptive)**

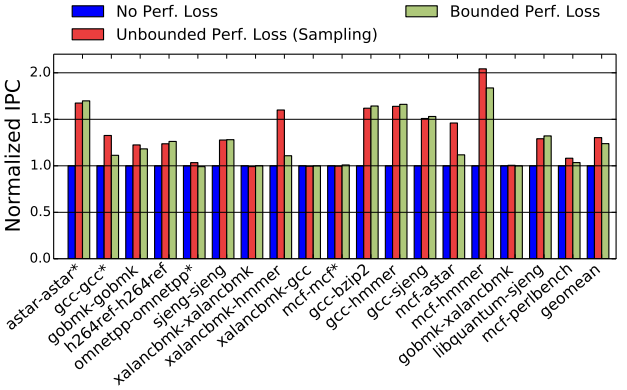


**Figure 6: Total throughput of NPL, UPL (Sampling), BPL (Adaptive)**

performance. Therefore, the scheme actually maximizes the entire throughput under a 95% performance limitation on the primary thread.

## 4.2 Results for Adaptive Cache Partitioning

Figure 3 compares the primary thread performance losses of No Performance Loss (NPL), Unbounded Performance Loss (UPL) and Bounded Performance Loss (BPL). In the case of NPL, all L1 data cache ways are assigned exclusively to the primary thread. All normalization in this paper is with respect to corresponding performance characteristics of NPL. For UPL, the L1 data cache is directly shared between two threads. BPL employs the adaptive cache partitioning scheme in the data cache. Compared with UPL, the adaptive scheme limits the primary thread performance losses of all the workloads to be lower than 5% and decrease the geometric mean performance loss to 3.3%.

The total throughputs of NPL, UPL and BPL are shown in Figure 4. Because the adaptive cache partitioning scheme in BPL should consider the primary thread performance limitation, it cannot utilize all the opportunities to maximize the total throughput and therefore leads to a loss on the entire performance. On average, the normalized throughput decreases from 1.31 to 1.24 with the adaptive cache partitioning scheme employed instead of directly sharing the data cache.

For some workloads, the primary thread performances almost do not decrease in BPL for two different reasons. The first reason is that there is no opportunity for cache sharing in most instruction phases. Directly sharing the data cache or assigning cache ways exclusively to the secondary thread will dramatically decrease the primary thread performance. For example, for the workload xalancbmk-gcc, the performance of xalancbmk is decreased by 18% in UPL, which illustrates that sharing data cache may leads to a significant performance loss for xalancbmk. Therefore, xalancbmk has no performance loss in BPL due to the first reason. The second reason is that the two benchmarks in the same workload do not have much cache contention with each other (e.g., libquantum-sjeng). In this case, the primary thread performance will not decrease with the data cache shared and the adaptive cache partitioning scheme actually has no impact on the performances of both threads.

## 4.3 Adaptive and Sampling Cache Partitioning Schemes

In addition to the adaptive cache partitioning scheme, an sampling scheme is also discussed in this paper.

The sampling cache partitioning scheme selects the cache partitioning modes for every instruction phase by briefly sampling the performance with different modes. For every instruction phase with 10 million instructions from the primary thread, the scheme executes the first 100K instruc-

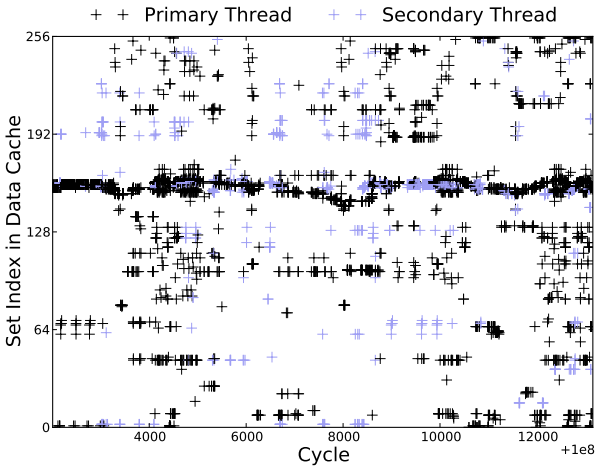


**Figure 7: Sample Case of gcc-gcc**

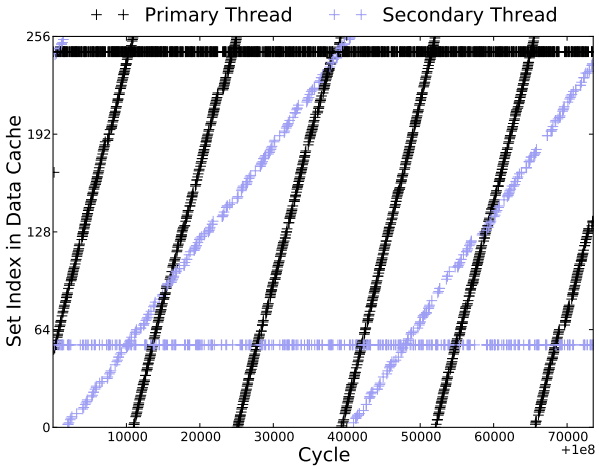


**Figure 8: Sample Case of libquantum-libquantum**

tions under all the cache partitioning modes including directly sharing the data cache. The mode that maximizes the total throughput of the sampling phase is then adopted for the entire instruction phase.

Figure 5 and Figure 6 show the primary thread performances and total throughputs of the sampling scheme. Because the sampling scheme actually adds no performance loss limitation on the primary thread, it is labeled as UPL (Sampling) in those figures. The results for the sampling scheme are similar to that with a directly shared data cache. Although it has a higher overall throughput, the sampling scheme still suffers from the performance loss of the primary thread.

## 4.4 Case Analysis

In this section, we give some example cases to explain in what situations our adaptive cache partitioning scheme works.

Figure 7 and Figure 8 plot the L1 data cache accesses of the primary and secondary threads in a single short time slot. The y axis is the set index of every data cache access instead of the memory address.

Figure 7 shows the cache accesses with workload gcc*-gcc*. The cache space accessed by the two benchmarks overlap substantially, which means there is severe cache contention between the primary and the secondary threads. Therefore, cache partitioning can help decrease the cache contention and improve the total throughput. Especially, the primary thread can avoid a significant performance loss with the protection given by our adaptive scheme. Two benchmarks both have a small memory set but a high cache reuse rate. Base on Table 1, the priorities of the two benchmarks are maintained. However, considering the limitation on the primary thread performance loss, the primary actually get all the cache ways in the data cache in this time slot.

The cache accesses of workload libquantum-libquantum are plotted in Figure 8. Both benchmarks have a similar cache access behavior which consists of two different access patterns. The first pattern keeps accessing the next cache block while the second one repeatedly accesses the same block. For the first pattern, the cache line loaded will not be reused in the near future and do not need to be saved in the data cache. Only the cache line that is repeatedly accessed in the second pattern should be saved and will be reused. Therefore, both benchmarks have a large memory set and a low cache reuse rate. As shown in Table 1, the priorities of both threads will decrease and the cache ways assigned exclusively to them may become directly shared. In real simulation, as predicted, the entire data cache is directly shared between the primary and secondary threads in this execution phase.

# 5. CONCLUSION

Traditional cache partitioning mechanisms focus on the last level cache and all threads are handled equivalently. However, in tightly-coupled architectures like Composite Cores, heterogeneous cores share resources up to the L1 caches, causing performance to suffer from L1-level cache contention. Moreover, two threads within a Composite Core should be considered differently due to the inherent heterogeneity. The primary thread should be protected from a significant performance loss.

This paper proposes an adaptive cache partitioning scheme for L1-level caches of a Composite Core. Cache capacities are dynamically resized every fixed-length instruction phase based on the memory set size and the cache reuse rate. Considering the inside heterogeneity, the primary thread tend to have a higher priority compared with the secondary thread. This scheme also provides a 5% limitation on the performance loss of the primary thread for performance protection.

Experiments in this paper are configured in an oracle way and only local performance information is used for cache capacities resizing. However, with history performance information taken into consideration for the adaptive cache partitioning scheme, the length of the instruction phase is expected to decrease, which can help expose more opportunities to improve the overall performance. Exploring the hardware implementation of the adaptive cache partitioning scheme with a finer granularity is an essential part of our future work.

# 6. REFERENCES

[1] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, “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*, pp. 81–92, IEEE, 2003.

[2] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 317–328, IEEE Computer Society, 2012.

[3] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 57–68, ACM, 2011.

[4] M. K. Qureshi and Y. N. Patt, "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*, pp. 423–432, IEEE Computer Society, 2006.

[5] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (prism)," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 428–439, IEEE Computer Society, 2012.

[6] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 356–367, IEEE, 2014.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.