Premier: A Concurrency-Aware Pseudo-Partitioning Framework for Shared Last-Level Cache

Xiaoyang Lu, Rujia Wang, Xian-He Sun
Department of Compute Science, Illinois Institute of Technology, Chicago, IL
xlu40@hawk.iit.edu, rwang67@iit.edu, sun@iit.edu

Abstract—**As the number of on-chip cores and application demands increase, efficient management of shared cache resources becomes imperative. Cache partitioning techniques have been studied for decades to reduce interference between applications in a shared cache and provide performance and fairness guarantees. However, there are few studies on how concurrent memory accesses affect the effectiveness of partitioning. When concurrent memory requests exist, cache miss does not reflect concurrency overlapping well. In this work, we first introduce pure misses per kilo instructions (PMPKI), a metric that quantifies the cache efficiency considering concurrent access activities. Then we propose Premier, a dynamically adaptive concurrency-aware cache pseudo-partitioning framework. Premier provides insertion and promotion policies based on PMPKI curves to achieve the benefits of cache partitioning. Finally, our evaluation of various workloads shows that Premier outperforms state-of-the-art cache partitioning schemes in terms of performance and fairness. In an 8-core system, Premier achieves 15.45% higher system performance and 10.91% better fairness than the UCP scheme.**

I. Introduction

In most multi-core systems, applications running on different cores share the last-level cache (LLC). As the number of cores on the chip increases, applications increasingly compete for the shared cache, which is detrimental to the overall system performance. As a result, it is critical to manage the shared cache to achieve high performance and fairness. Cache partitioning is an effective method to manage cache capacity per core and enforce access isolation between different workloads, thus mitigating contention and interference in shared LLCs.

Due to historical reasons, conventional cache partitioning schemes are designed to reduce cache misses, which may or may not be the best for concurrent cache memory accesses. Data access concurrency and overlapping are common in modern computing systems. In such cases, some cache misses occur concurrently with other hits (hit-miss overlapping), and the penalty of the misses could be reduced or hidden. As a result, classifying miss types may lead to a better understanding of miss penalty and a better optimized system performance.

In this work, we first introduce the concept and a formal definition of *Pure Misses Per Kilo Instructions (PMPKI)*. Unlike the classical misses per kilo instructions metric (MPKI), which only focuses on data locality, PMPKI takes into account overlapping in concurrent memory systems and reflects the number of *pure misses* (§II-A) that hurt the performance most. Next, we present Premier, a concurrency-aware shared cache management framework that takes both data locality and concurrency into account. Based on PMPKI curves, Premier provides insertion and promotion policies for each application to achieve efficient pseudo-partitioning. Our experimental results show that Premier outperforms state-of-the-art cache partitioning schemes in both performance and fairness.

II. Background and Motivation

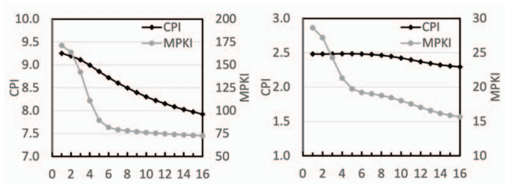
A. Concurrent Cache Accesses

Concurrent data accesses provide overlapping [10], which helps hide data access latency. At the **same cache level**, when the cache miss-access cycles overlap with the hit-access cycles, the cache miss penalty can be hidden because hit accesses continue to feed data to the processor [4]. Note that since each core has its own workload, memory accesses from different cores are not related. Only the overlapping of accesses from the **same core** is considered meaningful.

The term *Pure Miss* was introduced to identify the misses that are more harmful to performance when considering data access concurrency. In multi-core systems, pure miss in the shared cache is the miss access that contains at least one miss-access cycle, which does not have any hit access activity from the **same core** to overlap. Reducing the number of pure misses has proven to be an effective way to improve the overall memory system performance [5].

B. Cache Partitioning

Strict partitioning: Strict partitioning schemes in set-associative caches are typically implemented through way-partitioning, which provides each application with exclusive ownership of a specific partition. Qureshi and Patt proposed utility-based cache partitioning (UCP) [8]. UCP uses miss curves to determine partitioning decisions, which capture the core's misses for each possible partition size. Subramanian et al. proposed ASM cache partitioning [9]. ASM partitions the shared cache to achieve minimizing slowdown. However, to estimate the slowdown of the applications, the scheduler of the memory controller needs to be modified, which may negatively impact performance. Although strict partitioning schemes are straightforward, they may lead to low cache utilization [7].
Pseudo-partitioning: Pseudo-partitioning techniques implicitly partition the cache by managing the cache insertion and promotion policies. Xie and Loh proposed PIPP [11], which uses UCP's monitoring circuit to determine the insertion points for all new incoming lines from each core. PIPP only promotes the cache hit line by a single position with a certain probability when a cache line is hit. Kaseridis et al. proposed MCFQ [3],



(a) 605.mcf (b) 649.fotonik3d

Fig. 1: MPKI and CPI for SPEC CPU2017 benchmarks as the cache size is varied. (The *x-axis* is the number of ways allocated from a 16-way 2MB L3 cache to this workload.)

an MLP-aware pseudo-partitioning scheme. However, MCFQ does not realize that the miss penalty can be hidden when miss accesses overlap with hit accesses. Therefore, MCFQ analyzes concurrent memory accesses in a coarse-grained manner.

C. Motivation

Current cache partitioning schemes mainly aim at reducing the absolute number of cache misses and assume there is a high correlation between miss reduction and performance improvement [8]. Figure 1 demonstrates when concurrent memory requests exist, the correlation between the saved misses by additional cache capacity and the overall system performance is weak. For `605.mcf`, the number of misses tends to stabilize when the way is allocated exceeds 6. However, as the allocated cache increases, the CPI of the `605.mcf` continues to decrease. For `649.fotonik3d`, as the number of allocated ways increases from 1 to 5, the number of misses is significantly decreased. However, the CPI of `649.fotonik3d` stays constant from 1 way to 5 ways. Therefore, by monitoring the missing curves, cache partitioning schemes with the goal of reducing the total number of misses cannot ensure the highest performance gains. We are motivated to design a new cache partitioning scheme with a different performance optimization goal by considering the cache misses that harm the performance the most.

III. Pure Misses Per Kilo Instructions (PMPKI)

A. Definition and Measurement

We first introduce *PMPKI* to quantify the cache efficiency of a program in concurrent access activities. Different from the definition of MPKI, which relies on the ratio of miss accesses to evaluate the cache performance, PMPKI focuses on quantifying the ratio of *pure misses* (§II-A). PMPKI is definded as the number of pure misses per kilo instructions over a given time interval:

$$\text{Pure Misses Per Kilo Instructions} = 1000 \times \frac{\text{Num. of } \textbf{Pure Misses}}{\text{Num. of Total Instructions}}$$

B. Accuracy of PMPKI Metric

To verify the correctness of the PMPKI metric, we first measure the L3 PMPKI, L3 MPKI, and CPI for 20 evaluated workloads from SPEC CPU 2017 benchmark suite [2] in single-core configurations as the L3 cache size is varied. Then for each workload, we show the correlation (r) of PMPKI-CPI and MPKI-CPI. Figure 2 indicates that compare

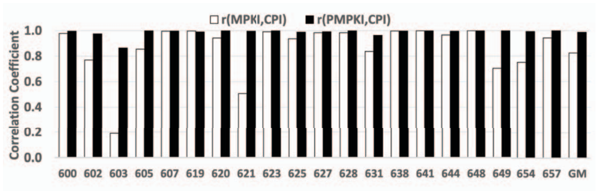


Fig. 2: Correlation coefficient analysis.

to MPKI, PMPKI shows a much higher positive correlation with CPI. For all workloads, the majority of r(PMPKI, CPI) are more than 0.99. The geometric mean of r(PMPKI, CPI)s is around 0.99, which is much larger than the geometric mean of r(MPKI, CPI)s. The strong correlation between PMPKI and CPI shows that PMPKI has advantages in capturing the concurrency/locality combined characteristics of modern memory systems.

C. Classify Workloads with PMPKI

The PMPKI curves also capture the sensitivity of the application performance with different cache sizes. We can classify workloads into different categories by directly monitoring the runtime PMPKI. **Cache-insensitive** applications are characterized by the fact that their PMPKI hardly changes as the cache size increases. The PMPKI of **cache-sensitive** applications continues to decrease as the cache size increases. **Cache-fitting** applications are also sensitive to allocated cache size. These applications benefit from the additional cache capacity until they are allocated enough cache space to fit their working sets. An increase in cache resources beyond their ideal capacity hardly further reduces pure misses.

IV. Premier: A Concurrency-Aware Pseudo-Partition Framework

A. Design Overview

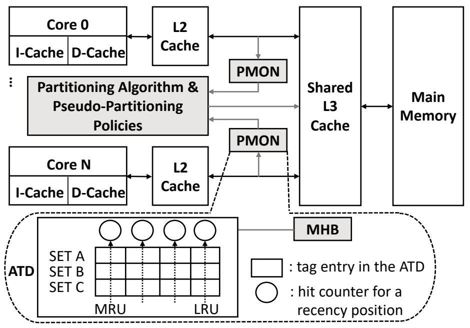


Fig. 3: Block diagram of Premier.

Figure 3 shows the overview of the Premier framework. The grey shaded modules are designs we added to a typical multi-core architecture. First, each core has a PMPKI monitoring circuit (PMON) to *estimate* the number of pure misses for each core when allocated in all possible cache partition sizes (in terms of cache ways) without interfering other running applications (§IV-B). Second, the applications are classified as cache-insensitive, cache-sensitive, or cache-fitting (§IV-C). Then, the partitioning algorithm utilizes the PMPKI curves estimated by PMONs to determine the cache size allocated to each core to minimize the number of pure misses in the system (§IV-D). Finally, Premier uses the pseudo-partitioning technique to sidestep the limitation of strict partitioning. At the end of each period (16K LLC misses), based on the category of each application provided by PMONs and the partitioning plan provided by the partitioning algorithm, Premier dynamically decides the insertion policy for cache misses and the promotion policy for hit accesses (§IV-E).

B. PMPKI Monitor (PMON)

To estimate the PMPKI curve when different numbers of ways are assigned to applications, an auxiliary tag directory (ATD) [8], [11] is assigned to each core, tracking the state of the cache if the core has exclusive access to the shared cache. The ATD has the same associativity as the main tag directory of the shared cache. Based on the stack property, a hit access at the *i-th* most recent position in the LRU stack indicates that the hit will be converted to a miss in the cache with less than i ways (for the same set count). Hit counters are assigned to each recency position ranging from MRU to LRU. By counting the number of hits corresponding to the LRU stack positions, a single ATD can provide the hit and miss information for all possible partition sizes at once. Each ATD is also attached with a miss holding buffer (MHB) which has the same entries of MSHR, to simulate the functions of MSHR. At the end of each period, PMON further estimates the number of pure miss accesses based on the hit/miss information provided by ATD, MHB, and the cycle information of each access.

C. Application Classification

Assuming there is an N-way set-associative shared cache, to reduce the computational complexity, Premier classifies applications based on the PMPKI value when the application is assigned only 1 cache way, $N-1$ ways, and N ways (noted as PMPKI$_1$, PMPKI$_{N\text{-}1}$ and PMPKI$_N$ respectively). If the ratio of PMPKI$_N$ and PMPKI$_1$ of an application is greater than a threshold T_{insen}, we consider the application is cache-insensitive. If an application is not cache-insensitive, and the difference between PMPKI$_{N\text{-}1}$ and PMPKI$_N$ is greater than a threshold T_{sen}, it is characterized as cache-sensitive. The remaining applications are classified as cache-fitting. Based on the analysis of the SPEC CPU 2017 benchmarks we evaluated, T_{insen} is set to 0.95 and T_{sen} is set to 0.1.

D. Partitioning Algorithm

Once PMON has completed the computation of the PMPKI curve for each application, Premier uses the PMPKI curves to feed into the *Lookahead algorithm* [8]. Due to the high correlation between PMPKI and performance, the lookahead algorithm is used to calculate the ideal partition cache sizes for each application online, intending to minimize the total number of pure misses incurred by all applications in the shared cache. The partitioning plan provided by the lookahead algorithm for k cores can be denoted as $\Omega=\{\omega_0, \omega_1, \ldots, \omega_{k-1}\}$ and $\sum_{i=0}^{k-1} \omega_i = N$, where N is the associativity of the shared cache.

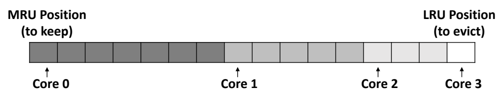


Fig. 4: The insertion positions for four applications.

TABLE I: Simulated system configurations

Processor	2 to 8 cores, 4GHz, 8-issue width, 256-entry ROB
L1 Cache	private, split 32KB I/D-cache/core, 64B line, 8-way, 4-cycle latency, 8-entry MSHR, LRU
L2 Cache	private, 256KB/core, 64B line. 8-way, 10-cycle latency, 32-entry MSHR, LRU
L3 Cache (LLC)	shared, 2MB/core, 64B line, 16-way, 20-cycle latency, 64-entry MSHR
DRAM	8GB 2 channels, 64-bit channel, 2400MT/s, tRP=15ns, tRCD=15ns, tCAS=12.5ns

E. Pseudo-Partitioning Policies

Insertion policy: Premier first assigns priorities to applications based on how sensitive they are to cache size. Premier assigns the lowest priority to cache-insensitive applications and provides the highest priority to cache-sensitive applications. If there are multiple applications in the same category, the priority between these applications is determined according to the PMPKI$_1$ of each application. Then, Premier combines the priority of the applications with the partition sizes calculated by the lookahead algorithm to determine the insertion point for each application. Figure 4 illustrates the insertion positions for a 16-way cache shared between four cores. Suppose the target partitioning plan is Ω={7, 5, 3, 1}; core$_0$ has the highest priority, followed by core$_1$ and core$_2$, and core$_3$ has the lowest priority. Premier only inserts new cache blocks near MRU positions for higher priority applications to ensure that higher priority applications get the cache capacity they require and encourage them to steal cache capacity from other applications. New cache blocks from lower priority applications are inserted close to the LRU position.
Hit-promotion policy: In order to improve the data locality, for cache-sensitive and cache-fitting applications, if a cache block receives a hit, Premier moves the cache block to the MRU position in the LRU stack. Otherwise, Premier only promotes the cache block to its insertion position.

V. Experimental Methodology

We use the ChampSim [1] simulator to evaluate Premier in multi-core systems. Table I describes the configuration used in our study. We select benchmarks randomly from the SPEC CPU 2017 benchmarks [2] to generate mixed-copy workloads as shown in Table II. We warm the cache for 50M instructions and measure the behavior of the next 200M instructions.

For each workload we evaluate the *throughput* (sum of IPCs, $\sum IPC_i$) and *fairness* (harmonic mean of normalized IPCs, $N/\sum(IPC_{i,alone}/IPC_i)$, where $IPC_{i,alone}$ is the IPC when the application executes in isolation under the ownership of all cache resources [6]). We select UCP [8] as the baseline for comparison. We further compare Premier against three state-of-the-art cache partitioning schemes: MCFQ [3], PIPP [11], and ASM [9].

TABLE II: Evaluated workloads

2-core	4-core	8-core
MIX 1: 603,623	MIX 1: 605,621,627,654	MIX 1: 607,619,620, 623,625,628,638,657
MIX 2: 603,654	MIX 2: 607,619,628,620	
MIX 3: 605,607	MIX 3: 621,605,602,603	MIX 2: 605,621,627, 649,654,620,623,628
MIX 4: 605,627	MIX 4: 619,623,602,603	
MIX 5: 607,619	MIX 5: 605,621,654,623	MIX 3: 605,621,654, 607,619,623,625,628
MIX 6: 619,623	MIX 6: 605,621,619,623	
MIX 7: 621,627	MIX 7: 621,619,623,620	MIX 4: 605,627,649, 654,619,628,602,603
MIX 8: 623,649	MIX 8: 621,649,623,603	
MIX 9: 627,654	MIX 9: 621,619,623,603	MIX 5: 605,654,607, 619,620,657,602,603
MIX 10: 649,654	MIX 10: 605,623,602,603	

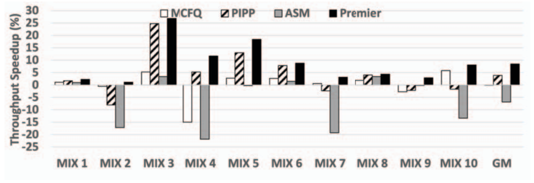


Fig. 5: Throughput speedup over UCP for 2-core workloads.

VI. Experimental Results

A. Performance Evaluation

Figure 5 shows that on the 2-core system, Premier outperforms existing schemes across the board, with a geometric mean speedup of 8.50% over UCP. For 4-core mixed workloads, Figure 6 shows Premier offers a speedup of 9.17% on average, an improvement of 7.62% over MCFQ, 4.95% over PIPP, 7.49% over ASM. Figure 7 shows that the Premier performance advantage comes from the fact that Premier significantly reduces LLC pure misses compared to the state-of-the-art schemes. Premier yields 4.13%, 3.32%, 2.09%, and 2.97% average pure miss reduction over UCP, MCFQ, PIPP, and ASM. Figure 8 shows that the advantage of Premier further increases on an 8-core system. Premier provides a 15.45% higher geometric mean throughput over the baseline UCP, 10.79% over MCFQ, 6.54% over PIPP, and 15.07% over ASM. When LLC cache size increases and the contention (core number) increases, we observe that Premier has a better opportunity to improve performance.

B. Fairness Evaluation

Figure 9 summarizes the fairness comparison as we increase the number of cores. Premier provides higher fairness than every state-of-the-art cache partitioning scheme in all configurations. Concurrency increases as the number of cores increases, and since Premier is concurrency-aware, the fairness advantage of Premier becomes greater. In the 8-core configuration, Premier achieves a fairness improvement over UCP by 10.91% on average.

VII. Conclusions

In this paper, we propose *pure miss per kilo instructions* (PMPKI), a metric that considers both data locality and concurrency. We present Premier, a concurrency-aware pseudo-partitioning framework based on monitoring the PMPKI of each application to provide the benefit of dynamic capacity allocation, adaptive insertion, and interference mitigation. Our evaluations across a wide variety of workloads and system configurations show that Premier is superior to the state-of-the-art cache partitioning schemes in terms of performance and fairness.

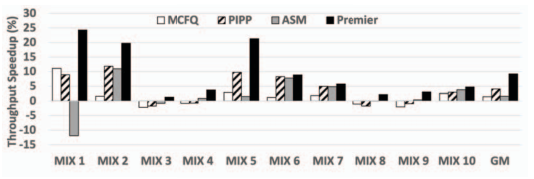


Fig. 6: Throughput speedup over UCP for 4-core workloads.

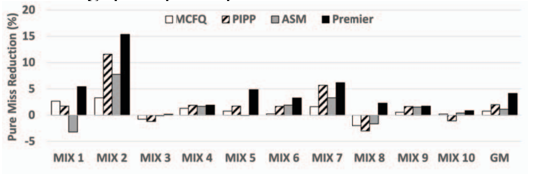


Fig. 7: Pure miss reduction over UCP for 4-core workloads.

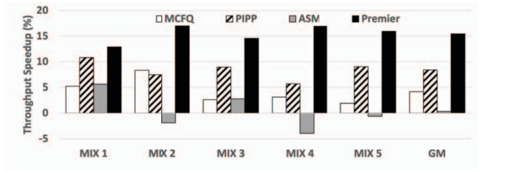


Fig. 8: Throughput speedup over UCP for 8-core workloads.

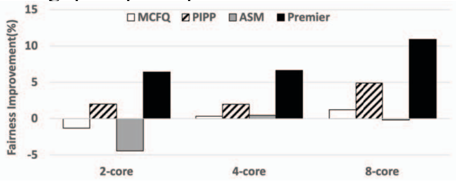


Fig. 9: Fairness improvement over UCP for 2, 4, 8 cores.

Acknowledgment

This research is supported in part by the National Science Foundation under Grants CCF-2029014, CCF-2008907, CNS-1730488, and by the NSF supported Chameleon testbed facility.

References

[1] The champsim simulator. https://github.com/ChampSim/ChampSim.
[2] Spec cpu2017 benchmark suite. http://www.spec.org/cpu2017/.
[3] D. Kaseridis, M. F. Iqbal, and L. K. John. Cache friendliness-aware management of shared last-level caches for high performance multi-core systems. *IEEE transactions on computers*, 63(4):874–887, 2013.
[4] J. Liu, P. Espina, and X.-H. Sun. A study on modeling and optimization of memory systems. *Journal of Computer Science and Technology*, 36(1):71–89, 2021.
[5] X. Lu, R. Wang, and X.-H. Sun. Apac: An accurate and adaptive prefetch framework with concurrent memory access analysis. In *ICCD-38*, 2020.
[6] K. Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in smt processors. In *ISPASS'01*, 2001.
[7] S. Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Computing Surveys (CSUR)*, 50(2):1–39, 2017.
[8] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO-39*, 2006.
[9] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *MICRO-48*, 2015.
[10] X.-H. Sun and D. Wang. Concurrent average memory access time. *Computer*, 47(5):74–80, 2013.
[11] Y. Xie and G. H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. 2009.