

CACHENCE: Fine-Grained Cache Partitioning in Both Time and Space

Liujia Li¹, Yuanlong Li², Yiming Yao¹, Jianyu Wu¹, Yi Fan¹, Jinhao Guo¹, Liren Zhu¹,
Jie Zhang¹, Xiaolin Wang¹, Yingwei Luo¹, Zhenlin Wang³, Diyu Zhou¹

¹Peking University, Beijing, China; ²EPFL, Lausanne, Switzerland; ³Michigan Technological University, Houghton, USA

Abstract

Cache partitioning mechanisms improve system performance by judiciously allocating cache space among diverse applications. We found that the effectiveness of cache partitioning can be significantly improved for modern applications through *fine-grained* partitioning in both time and space. However, existing software and hardware solutions are limited to only one dimension.

This paper presents CACHENCE, a novel cache partitioning mechanism that achieves *both* **timely** and **high-granularity** partitioning with two novel techniques. First, CACHENCE profiles cache access patterns by implementing a refined cache analytical model in hardware, enabling fine-grained miss-ratio curve profiling. The hardware-friendly optimization of the model provides high accuracy with *theoretical guarantees* in less than *tens of million cycles*, supporting timely partition while incurring only a small, about two kilobytes space overhead for each core. Second, CACHENCE leverages the characteristics of modern out-of-order processors and runtime *microarchitectural* statistics to propose a lightweight yet precise performance model, which supports a *thousand of kilobytes-level* cache partitioning. CACHENCE also proposes a special hardware cache partitioning tool, boosting the *hit rate* while minimizing *storage and latency overhead*. Our evaluation shows that, on a 16-core system, CACHENCE outperforms existing approaches by an average of 9.5% to 28.2%, and up to 48.8%.

1 Introduction

Judiciously partitioning the last level cache (LLC) among applications promises to improve overall performance [3, 5, 7, 14, 17, 18, 20, 26, 29, 31]. Modern cache partitioning mechanisms [7, 20, 29] improve over equal division with better partitioning decisions considering *time* and *space*. In terms of time, because modern applications exhibit phase behavior, modern mechanisms re-partition cache spaces among applications at regular time intervals. In terms of space, modern mechanisms monitor cache accesses and analyze how the performance of each application varies with different cache sizes. They then prioritize allocating caches to applications whose performance is more sensitive to cache size.

We argue that the partition mechanism should be **both** temporal and spatial fine-grained. Any absence of fine granularity on time or space can lead to significant performance degradation. As shown in Figure 1(a), modern applications are executing fast with the rapid advancement across the computing stack. The shorter execution

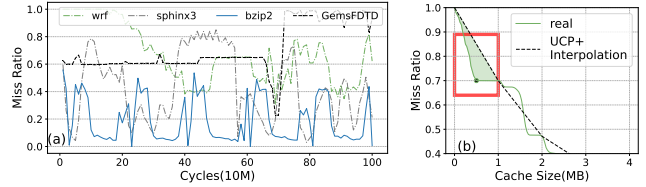


Figure 1: (a) Cache miss ratio over time for four representative SPEC CPU programs, showing drastic changes in cache access patterns over tens of millions of cycles. (b) Cache miss ratio for wrf with varying cache sizes during a random 20-million-cycle time slice. The dotted line is generated by applying interpolation to UCP, and the shaded area is the error with this approach (§2).

stage requires a timely partition. Figure 1(b) shows a small reduction in cache space often leads to a drastic increase in the cache miss ratio. As a result, a high-granularity spatial partition is a must.

However, existing cache partitioning mechanisms [7, 20, 29] cannot simultaneously achieve fine-grained partitioning in both time and space. Software approaches [7, 29] work on commodity hardware, achieving relatively fine-grained spatial partitioning. However, software prevents the partition mechanism from responding quickly to phase changes, resulting in coarse temporal granularity. Hardware partitioning approaches, for example UCP [20], enhance the underlying hardware with a specific design for the widely used set-associative caches. UCP achieves fine-grained temporal partitioning, but the hardware design is limited to coarse-grained spatial partitioning at the granularity of cache ways.

This paper presents CACHENCE, a novel hardware cache partitioning mechanism that achieves fine-grained partitioning in both time and space. The main design challenge in CACHENCE is maintaining *accuracy*. On the one hand, temporal granularity requires a timely and accurate awareness of phase changes. On the other hand, spatial granularity needs an accurate performance estimation to fully exploit the opportunity to maximize hit rate, since the penalty of wrong decisions can be high. Meanwhile, it is necessary to keep storage and latency overheads as low as possible. As we detail below, CACHENCE overcomes this challenge using a set of novel and effective techniques.

CACHENCE improves AET [12], a recent advancement in cache analytical model which, however, lacks a hardware-friendly sampler design. AET takes the reuse time of memory access as the input, and it is unacceptable for hardware to record all memory accesses to obtain an accurate reuse time distribution. CACHENCE uses improved reservoir sampling in the AET model with a fixed buffer size for hardware implementation, achieving faster sampling speed and enabling timely cache partitioning.

AET also lacks an evaluation of application performance and only provides cache hit rate. Cache hit rate and performance are not exactly the same. Allocating resources solely based on cache



This work is licensed under a Creative Commons Attribution 4.0 International License. DAC '26, Long Beach, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2254-7/2026/07

<https://doi.org/10.1145/3770743.3803974>

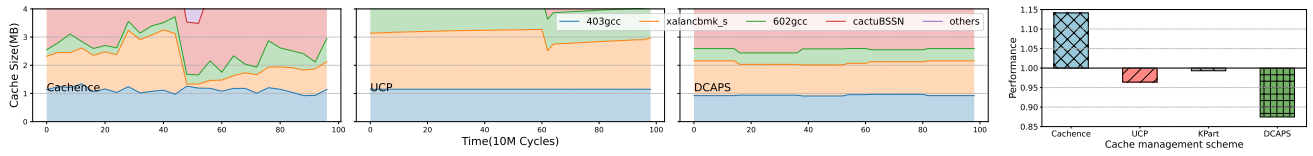


Figure 2: Cache partitioning changes over time with a mix of eight applications. Both DCAPS and UCP barely adjust the partitioning since 1) DCAPS suffers from a long adjustment interval; and 2) UCP adjusts space at the way granularity and thus miss the opportunities to improve.

hit rate may lead to suboptimal allocation and fail to fully exploit high-granularity spatial partitioning. To bridge the gap between allocated cache sizes and performance, CACHENCE proposes a light-weight performance model. The model is based on the observation that, in modern processors, a cache miss hurts performance by stalling the initiator load, thereby preventing younger dependent instructions in the re-order buffer (ROB) from execution. As a result, the performance model estimates the performance reduction due to a cache miss by using the ratio of (i) the number of instructions in ROB and (ii) the number of cache misses in the miss status holding register (MSHR). CACHENCE then tackles the cache partitioning problem with *dynamic programming* [15]. Given the input from our performance model, it *finds the optimal partitioning*, departing from the heuristic algorithms proposed by UCP.

To reduce overhead, CACHENCE implemented a specially designed hardware cache allocation tool for modern Intel and ARM platforms. Experiments on a 16-core system show that CACHENCE outperforms state-of-the-art mechanisms by an average of 9.5% to 28.2%, and up to 48.8%. CACHENCE can re-partition the cache in less than 200K cycles, which is negligible compared to the target adjusting interval (*i.e.*, tens of millions of cycles), while only incurring less than 1% storage overhead compared to L2 caches.

In summary, we make the following contributions:

- We demonstrate the necessity of fine-grained cache partitioning in both time and space, identifying optimization opportunities through extensive analysis of the SPEC CPU benchmarks [1, 4, 11].
- We introduce two effective and hardware-friendly algorithms and establish theoretical effectiveness: (1) a quick sampling algorithm that refines reservoir sampling which enables timely profiling and (2) an accurate performance estimation model that guides the high-granularity spatial partition.
- We design and implement CACHENCE, a cache partitioning approach that realizes the aforementioned new algorithms in hardware with minimal overhead.

2 Background and Motivation

Cache Partitioning Mechanisms A cache partitioning mechanism works in repetitive iterations where each iteration consists of two phases: the profiling phase and the partitioning phase. In the profiling phase, the mechanism collects information on cache access and calculates the Miss Ratio Curve (MRC) [19] for each application. Some mechanisms include a performance model that builds on the MRC to estimate, given the allocated cache size, the performance of an application. In the partitioning phase, the mechanism decides how to partition the cache among applications to maximize performance. This partitioning is based on predictions from the performance model (or the MRC), with more cache spaces allocated to applications that benefit most from larger cache sizes.

Why Fine-grained Cache Partitioning To motivate fine-grained cache partitioning, we analyzed the cache access patterns for all workloads in SPEC CPU 2006 [11] and 2017 [4] with ChampSim [8] (using the same experimental setup as shown in §5).

From Figures 1(a) and 1(b), we observe that cache partitioning must be fine-grained with respect to *time* and *space* to improve system performance. Figure 2 compares existing cache partitioning mechanisms on a representative workload mix. As shown in Figure 2, CACHENCE performs more timely and high-granularity partitioning, thereby outperforming others by 15%. To be specific, CACHENCE adjusts the cache partitioning 10 times in 200M cycles and CACHENCE allocates cache space at the granularity of 256KB. This allows workloads to have minimal cache immediately after it changes to a cache-insensitive phase. Figure 2 shows that xalancbmk is cache-insensitive during the period between 420M and 450M cycles. CACHENCE is able to respond timely and reduce the cache size allocated to xalancbmk immediately. The high-granularity allows low occupancy down to 256KB, sparing more space for cache-sensitive applications. In contrast, UCP has a fine-grained partitioning in time, but failed to decrease xalancbmk’s occupancy due to its coarse granularity. DCAPS allows fine-grained partitioning in space, but cannot respond quickly enough and misses the cache-insensitive phase of xalancbmk.

Existing Cache Partitioning Approaches Most existing software approaches use Intel CAT [6] for cache partitioning, which supports only the space partitioning granularity of a cache way. KPART [7] and DCAPS [29] achieve fine-grained partitioning in space by allocating a cache way to a cluster that consists of multiple applications. However, these approaches are still sub-optimal, since they neglect cache contention within a cluster. Moreover, their phase needs to be as long as hundreds of million cycles to obtain MRC. DCAPS uses Precise Event-Based Sampling (PEBS) [9] to sample cache accesses. KPART obtains the MRC by gradually reducing allocated cache space for the application and reading the performance counter.

The inherent limitations of software motivate the use of hardware methods. UCP [20] is a low-overhead, high-performance, hardware mechanism to partition shared caches, but still partitioning at the cache way granularity. Specifically, for each application, UCP enumerates the remaining number of unallocated cache ways and calculates the ratio of the increase in cache hits to the increase in cache sizes. With modern caches, this granularity is at least an order of magnitude larger than the granularity CACHENCE targets.

UCP also suffers from low accuracy for the lack of a performance model; it simply assumes that minimizing the total number of LLC misses across all applications leads to the best performance. However, this assumption is inaccurate because different applications are affected by cache misses to different degrees. What’s more, UCP

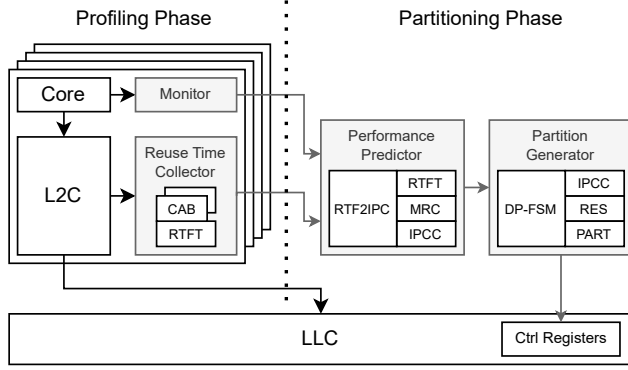


Figure 3: Overview of CACHENCE

uses a heuristic algorithm to find the cache partitioning. The above shortcomings prevent UCP from providing spatial fine-grained partition, although the hardware supports timely phase awareness.

3 Design

3.1 Design Goals

We design CACHENCE to meet the following design goals.

- **Timeliness.** CACHENCE should react to phase changes promptly to keep up with the evolving phase behavior of modern applications.
- **Accurate performance prediction.** The performance prediction in CACHENCE must be accurate enough to support a high-granularity cache partition.
- **Hardware-friendliness.** CACHENCE must be hardware-friendly in order to fully exploit the performance benefits offered by the hardware.
- **Low overhead.** CACHENCE must have minimal storage and latency overhead.

3.2 Overview

Figure 3 shows the overall architecture of CACHENCE. To provide real-time cache status for timely responses, each L2 cache is paired with a reuse time collector (Collector) that continuously monitors all L2 cache misses off the critical path and profiles the cache access patterns of that core. Being specially designed, the Collector is hardware-friendly and provides quick data collection.

For accurate performance prediction, a simple per-core monitor module collects the relevant information for performance estimation, including the number of outstanding requests in the core’s miss status holding register (MSHR) and re-order buffer (ROB). A global performance predictor (Predictor) calculates the estimated IPC for each core. Predictor first converts the cache access patterns received from Collector to MRC and then uses the information from the monitor to convert MRC to estimated IPC.

Finally, another global module partition generator (Partitioner), which ensures the low overhead of CACHENCE, generates an accurate final partitioning plan from the estimated IPCs of each core. Partitioner also implements a specially designed hardware partitioning tool, which dispatches the partitioning plan to the control registers inside the LLC controller.

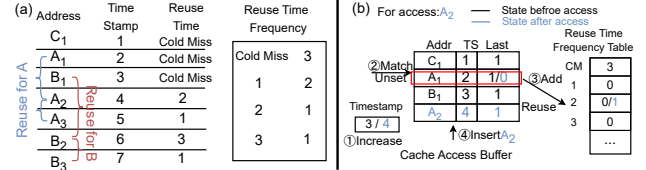


Figure 4: (a) Examples of reuse time and its frequency. A_i: the i -th access to address A. (b) Status of the reuse time collector after adding A₂ before adding A₃ in the example stream shown in the left.

3.3 Reuse Time Collector

The reuse time collector (Collector) monitors the L2 cache misses to collect information about the core’s memory access pattern. It collects the *reuse time* and its *frequency* for the cache accesses, which are key inputs to the AET model to generate MRC.

Reuse time and its frequency. The reuse time denotes the number of *cache accesses* between the current access and the prior one that accesses the same memory address. If there is no prior access to the same address, it is a cold miss and should also be counted. A shorter reuse time reflects a shorter interval between subsequent accesses, and the access is more likely to hit the cache even with a small cache size, and vice versa. The AET model collects reuse time, calculates reuse the time distribution, and predicts the MRC with a mathematical model. We use the “reuse-time frequency” to represent the distribution of all reuse times in a given stream of cache accesses (including cold misses). Figure 4(a) presents an illustrative example of reuse time and reuse time frequency.

Figure 4(b) illustrates the high-level baseline design of the Collector. The cache access buffer (Access Buffer) stores the cache access stream, the reuse time frequency table (Frequency Table) maintains the reuse time frequencies, and a Timestamp Counter increments for each cache access. Each Access Buffer entry contains the target memory address, a timestamp, and a last bit. The last bit indicates whether this entry is the last access to the target address, which is also a *valid* bit for sampling, since storing all memory accesses is impossible.

The challenge of sampling in collecting reuse time is that computing the reuse time requires the timestamps of two consecutive accesses to the same address. However, either of the two accesses may not be sampled, causing the computed reuse time to deviate from the true value. For example, there are 3 consecutive accesses A₁, A₂, and A₃ to the same address A at times T₁, T₂, and T₃. A₁ and A₃ are sampled. Unaware of A₂, the Collector will find the last access to address A is A₁ and regard the reuse time of A₃ as T₃ – T₁, which is incorrect. CACHENCE uses the last bit to represent whether the entry is available for calculating. When A₂ arrives, although it is not sampled, the Collector will unset the last bit of A₁, preventing the incorrect reuse time result for A₃.

More specifically, when the Collector detects a L2 cache miss, it increments the Timestamp Counter and looks up the entire Access Buffer to see if there is an entry with the same address and the last bit set. If so, the Collector calculates the reuse time of this access as the difference between the current Timestamp Counter and the timestamp of the matching Access Buffer entry. If there is no entry with the same address, or the last bit of the entry for that address is not set, the Collector regards this access as

a cold miss according to AET model and assigns infinity as the reuse time value. Later, the Collector increments the counter in the Frequency Table corresponding to the reuse time and creates a new Access Buffer entry for this access, with the last bit set and timestamp being the current Timestamp Counter. Figure 4(b) shows an example of the aforementioned process.

High-accuracy hardware-friendly sampling with bounded buffer size. In an ideal design, the Access Buffer should store all cache accesses to accurately capture the reuse time distribution, but this is infeasible in hardware practice. Following prior cache partitioning approaches [7, 20, 29], CACHENCE samples cache accesses to minimize storage and performance overhead. Unlike prior approaches, CACHENCE utilizes a more accurate and hardware-friendly sampling algorithm that provides a *theoretical guarantee* and only requires a *fixed* amount of buffer size, making it practical for hardware implementation, as we detail below.

Reservoir sampling assumes a buffer with a fixed size of E . The N -th cache access in the stream has a probability of E/N being sampled (*i.e.*, added to the buffer). If the buffer is full and a new access is sampled, a random entry is evicted. Reservoir sampling is highly accurate (*i.e.*, minimal errors in the collected reuse time frequency) since it mathematically guarantees **even sampling**: at the end of a cache access trace of length L , each access remains in the buffer with an equal probability of E/L .

Optimization: Minimizing performance overhead by exploiting composability. We further reduce the overhead of CACHENCE with the insight that the reuse time frequency is composable [12]. Given a set-associative cache, the reuse time frequency of a cache access stream can be obtained from the reuse time frequency of its individual sets. The composability of the AET model states that for two applications with their own reuse time frequencies and access rates, one can weight the two reuse time frequencies by their access rates and sum them up to get the overall reuse time frequency of the combined access stream. The new reuse time frequency accurately reveals how two applications work together in a shared cache.

Therefore, to generate an application’s reuse time frequency in a set-associative cache, CACHENCE treats each set as an independent application with a small Access Buffer. CACHENCE samples multiple sets to generate their reuse time frequencies separately and leverages the composability of AET to generate the overall reuse time frequency. To avoid calculation for access rates of each set, CACHENCE groups multiple sets into a sampling group with a shared Access Buffer taking them as similar access rates.

Calculating MRC. We follow the standard process described in the AET model [12] to calculate the MRC from the Frequency Table by Predictor. First, from the Frequency Table, the Predictor calculates the complementary cumulative distribution function (CCDF) of the reuse time. Then, for each cache segment, the Predictor sums the CCDF values one by one and searches for an index whose sum exceeds a corresponding predefined fixed threshold in AET. The indexed value in the CCDF is the miss ratio for the cache segment. We optimize the above process by first scaling CCDF (and the threshold) up by the number of L2 misses, thereby converting the floating-values in CCDF to integer values, which are easier for hardware to handle. Later, to maintain correctness, the Predictor divides the misses by the number of L2 misses.

3.4 Accurate Performance Prediction

IPC does not change with miss rates in a linear way [10]. Therefore, CACHENCE includes a performance model that converts miss rates generated by §3.3 into IPC estimations. With the predicted IPC, CACHENCE can then determine the optimal cache partition scheme.

Unlike some prior work [7, 20], CACHENCE uses a hardware-friendly performance model that requires minimal hardware modification. Our performance is obtained from the formula [10].

$$CPI = CPI_{Execution} + \frac{\#Memory\ Access}{\#Instructions} * Miss\ Rate * Penalty$$

LLC misses will stall CPU execution for a period, so we simplify the model by taking the average memory access penalty as the *Penalty* and LLC miss rate as the *Miss Rate*. In the profiling phase, CACHENCE collects the number of executed instructions, LLC status, and the cumulative memory access penalty for each core (§3.5). LLC status includes the number of LLC accesses and misses. Then, with the miss rates of different cache sizes generated in §3.3, CACHENCE calculates the *CPI* with the formula above under a certain cache size. CACHENCE can estimate the target IPC in different cache sizes in the performance model.

CACHENCE’s cache partitioner is motivated by the observation that cache partitioning can be formalized as a *resource allocation problem*, a field widely studied in mathematics [15]. Specifically, the resource allocation problem assumes S units of resources (S cache segments in our case), N activities (N cores), where each activity is associated with a return function $r(i)$ (the IPC table), denoting the return with i units of resources. CACHENCE implements dynamic programming to find an allocation plan to maximize the sum of returns (the aggregate IPC).

3.5 Implementation and Practicality

Hardware Modification. CACHENCE requires a few hardware modifications to implement the performance model, following prior work [13, 20, 24]. First, CACHENCE needs a counter for each core to record committed instructions in the sampling period. Second, cachelines in the cache need to be extended with core IDs to identify the source application of each cache access, as suggested in ARM MPAM [16]. Each LLC core has two counter registers to count LLC accesses and misses during the sampling period. Finally, CACHENCE extends the MSHR of the LLC, adding a timestamp to each entry to cumulate the penalty. With the timestamp, CACHENCE calculates the memory access penalty of the LLC misses for each core and the average memory access penalty after the sampling period.

CACHENCE associates each core with an occupancy register and a target cache usage to enforce the partitioning plan [16, 18]. The occupancy register records the current cache usage of each core, while the target usage register holds the allocated cache size decided by the partition generator. Upon eviction, CACHENCE first compares the occupancy register of each core with the target usage register, checks whether it exceeds the target usage. Second, CACHENCE picks up the cache lines by core ID, whose occupancy register exceeds the target usage register, as eviction candidates. If there is no eviction candidate, CACHENCE passes all the cache lines to the replacement policy, as there is no partitioning violation. Otherwise, the filtered eviction candidates are passed to the replacement policy, which selects one cache line to evict. **Storage**

Processor	4/8/16 cores, 2GHz, 128-entry load queue, 352-entry ROB
L1 Cache	private, split 64KB I/D-cache/core, 64B line, 4-way, 2-cycle latency, LRU
L2 Cache	private, 512KB/core, 64B line, 8-way, 10-cycle latency, LRU
LLC Cache	shared, 2MB/core, 64B line, 16-way, 20-cycle latency, LRU
DRAM	2 channels, 3200MT/s, tPR 12.5ns, tRCD 12.5ns, tCAS 12.5ns

Table 1: Simulated system configuration

Overhead. CACHENCE leverages on-chip SRAMs to store various data structures. For brevity, we assume that the system contains N cores, sharing S LLC segments. CACHENCE consists of N per-core Collectors, one Predictor, and one Partitioner. We fix the number of Access Buffers in each Collector. For example, there are 16 entries in the Access Buffer, every 64 bits. The Frequency Table is overprovisioned with $2S$ entries to provide sufficient accuracy to the MRC calculation. Each Frequency Table entry holds a 32-bit counter, requiring $2S \times 32 = 64S$ bits. The global Predictor contains a CCDF table of the same size as the Frequency Table. It also contains the MRC and IPCC, each with S entries of two bytes. As an example, for a 16-core system with 64 LLC segments (16MB), CACHENCE requires about 34.5 KB storage in total, which is negligible compared to the LLC.

Calculation Overhead. We leverage a dedicated Predictor for performance prediction (§3.4) and a DP-FSM (dynamic programming finite-state machine) for partition scheme generation which accesses the SRAMs directly. The access latency of SRAM and the overhead of basic arithmetic operations are both assumed to be one clock cycle, according to the specifications of a modern microprocessor in [2]. For floating-point computation, FP16 is used as the numeric format, with addition/subtraction, multiplication, and division requiring 2, 5, and 20 cycles, respectively. In the partitioning phase (§3.4), the Predictor traverses the Frequency Table to calculate the CCDF and MRC, which incurs $4S$ integer additions and S floating-point divisions, along with $7S$ SRAM accesses. Later, to calculate the estimated IPC for every number of LLC segments allocated to the core, the Predictor performs one floating-point subtraction and one multiplication. In total, the performance predictor requires $40S$ cycles to compute all the estimated IPCs for each core. A dynamic programming will consume about $O(NS^2)$ time to find the optimal partitioning scheme, corresponding to $2S(S+1)$ cycles. For example, in a 16-core system with 64 segments, the performance prediction consumes 40K cycles, while the partition generation consumes another 131K cycles, which is negligible compared to the profiling stage (tens of millions of cycles).

4 Methodology

We implement CACHENCE in ChampSim [8], a cycle-accurate simulator that models the entire memory system. Table 1 shows the detailed configuration of the simulated system.

We evaluate CACHENCE using workloads from SPEC CPU 2006 [11] and 2017 [4] with DPC-3 [1] traces. Each workload is categorized as either cache-sensitive or cache-intensive, according to whether its IPC increases by at least 5% when the LLC capacity

	Cache sensitive		Cache intensive	
SPEC CPU 2006	bzip2	lbm	leslie3d	hmm
	gcc1	omnetpp	bwaves1	wrf
	gcc2	cactusADM	bwaves2	mcf
	zeusmp	sphinx3	GemsFDTD	milc
	astar	xalancbmk		
SPEC CPU 2017	mcf_s	cam4	bwaves_s1	gcc_s
	pop2	omnetpp_s	bwaves_s2	lbm_s
	wrf_s	roms	fotonik3d	
	xz	xalancbmk_s	cactuBSSN	

Table 2: Evaluated workloads

is quadrupled from 2MB to 8MB. Table 2 lists the workloads in both categories. To form proper workloads for a multi-core system with N cores, we create workload mixes, each comprising $N/2$ randomly picked cache-sensitive and $N/2$ cache-intensive workloads, and we run each workload on a separate core [7, 29]. In each experiment, we first simulate for 200M cycles to warm up the cache hierarchy, and then continue for another 2B cycles for measurement. Similar to DPC-3 [1], if the trace of a workload finishes early, we replay it until the simulation ends.

We select the LRU replacement policy without LLC partitioning as the baseline for performance comparison. We use the aggregated IPC from all applications $IPC_{sum} = \sum IPC_i$ as the main performance metric, which is common in the literature [20, 29]. We define the speedup achieved by a cache partitioning scheme as the ratio of its aggregated IPC over that of the baseline.

We compare CACHENCE with three state-of-the-art cache partitioning mechanisms, i.e., UCP [20], DCAPS [29], and KPART [7]. To minimize the impact of software profiling, for DCAPS and KPART, we allow them to obtain MRC from the UCP hardware during warm-up and apply their cache partitioning schemes during the main simulation to cover the overhead of their partitioning phase.

5 Evaluation

In this section, we first explore the design space of CACHENCE to obtain the best configuration for the following experiments. Later, we evaluate CACHENCE in comparison with other cache partitioning schemes to demonstrate CACHENCE’s performance benefits.

5.1 Design Space Exploration

CACHENCE has several parameters that can be tuned to achieve the best performance. The most important one is the time interval between two consecutive cache partitioning adjustments. Figure 5 shows the performance of CACHENCE across various time intervals, from 5M cycles used in UCP [20] to 100M cycles, as in software methods [7, 29]. Figure 5 shows that CACHENCE achieves the best performance when the adjustment interval is 40M cycles, 7% better than the baseline on average. For adjustment intervals smaller than 40M cycles (e.g., 5M cycles), the time is too short for applications to set up stable phases, which makes it hard to collect representative reuse time statistics and generate high-accurate MRCs for the application, thereby leading to suboptimal partitioning decisions. For larger adjustment intervals, applying partition decisions, even when stable, lags behind the changes in the application’s execution phase and cache access pattern, leading to lower performance. We

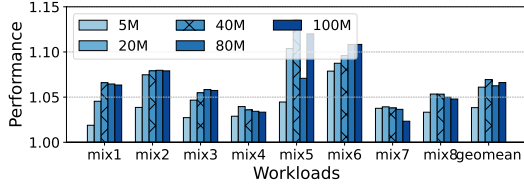


Figure 5: Performance with different time granularity.

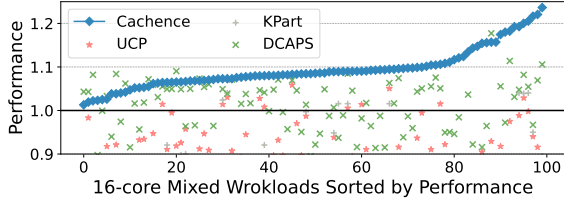


Figure 6: Performance for 16-core mixed workloads

therefore choose a 40M cycle as the optimal adjustment interval for the following experiments.

5.2 Comparison with State-of-the-Art

Figure 7 shows the performance of various cache partitioning schemes. In a 4-core system, on average, CACHENCE achieves about 6% performance improvement over the baseline due to its fine-grained partitioning scheme. DCAPS achieves a 4% performance improvement, while UCP and KPART even deteriorate the system performance. While KPART effectively isolates cache-sensitive applications from cache-intensive ones but clusters cache-sensitive applications together, which neglects the partitioning between cache-sensitive applications. DCAPS, on the other hand, achieves better performance with its empirical parameters.

The performance benefit of CACHENCE can be better illustrated with more cores. As shown in Figure 7, in an 8-core system, CACHENCE outperforms the baseline by 11.1%, while UCP, KPART, and DCAPS can barely improve performance in almost all the workload mixes. For UCP, because it relies on way partitioning when the core count approaches the LLC associativity, it fails to partition the cache, as it needs to reserve one way per core. DCAPS and KPART are limited by their overly coarse adjustment interval which fails to capture the fine-grained phase changes in the applications.

In a 16-core system, CACHENCE also achieves 9.1% performance boost over the baseline, while UCP, KPART, and DCAPS still strive to improve performance. In this case, UCP is degraded to allocate only one way per core, leading to a significant performance drop. While KPART suffers from its cluster method, which neither allocates cache space to cache-sensitive applications nor limits the cache space for cache-intensive. With increased workloads, the system's environment changes, but DCAPS's empirical parameters fail to adapt to, leading to a performance drop. Figure 6 further shows the performance improvement achieved by various cache partitioning schemes across all workload mixes sorted by the speedup of CACHENCE. It shows that CACHENCE outperforms other schemes in almost all workload mixes, with only 10 exceptions out of 100 mixes in which DCAPS achieves higher performance. For several mixes, CACHENCE even boosts their performance by more than 20%.

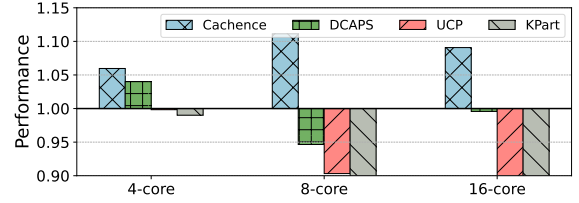


Figure 7: Performance for 4, 8, 16 cores.

6 Related works

This section presents other related work.

Vantage [22] and Futility Scaling [25] leverage the structure of zcache [21]. Zcache is an implementation of skew-associative cache [23], with one cache address map to different cache sets based on the hash of the address. Vantage and Futility Scaling use this property to partition the cache space among applications, and they proposed a mathematical model to decide the partitioning scheme.

The Floria [30] uses reverse engineering to understand the cache structure by deploying an interference program that frequently reads and writes to the target program's cache, restricting the cache usage to a specific range. However, this method can gather only basic information for a single target program at a time and requires deep knowledge of hardware mechanisms.

The footprint method [28] is a widely used method to calculate MRC. Xiang [27] divides the accesses into multiple intervals with a fixed length and then calculates the footprint, thereby generating the MRC for an application. Wang [26] uses footprint theory to analyze and quantify mixed program execution, proposing an adaptive bursty footprint sampling strategy to schedule mixed program execution scenarios; However, it needs a space to store the access information, which is not suitable for hardware implementation.

7 Conclusion

This paper presents CACHENCE, a novel hardware cache partitioning mechanism that simultaneously achieves fine-grained space (with a granularity of hundreds of kilobytes) and time partitioning (re-partitions every hundreds of million cycles). CACHENCE proposes three novel techniques that are highly accurate with minimal storage and space overhead. First, CACHENCE leverages the AET model to construct MRC by collecting reuse time distances in cache accesses with a bounded buffer size. In addition, CACHENCE refines the reservoir sampling algorithm by exploiting unused space to improve accuracy while proving that it still satisfies the equal-probability sampling property. Second, CACHENCE proposes a lightweight yet effective performance model that collects a few statistics from the ROB and MSHR. Finally, CACHENCE shows that dynamic programming can be used to find the optimal cache partitioning plan. Our extensive evaluation shows that CACHENCE significantly outperforms existing approaches in the majority of scenarios.

8 Acknowledgments

We thank the anonymous reviewers. The work is supported by the National Key R&D Program of China under Grant No. 2023YFB4502303, and by the National Science Foundation of China (Nos. 62372011, 92582118, 62032001). Diyu Zhou is the corresponding author.

References

- [1] 3rd data prefetching championship.
- [2] Micro processor high performance 32-bit mcu with dsp & fpu.
- [3] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. Optimal cache partition-sharing. In *2015 44th International Conference on Parallel Processing*, 2015.
- [4] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018.
- [5] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: QoS-aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, April 2019.
- [6] Intel Corporation. Intel® Resource Director Technology (Intel® RDT) Monitoring and Enforcement Primer. Technical report, Intel Corporation, 2016.
- [7] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
- [8] Nathan Guber, Gino Chacon, Lei Wang, Paul V Gratz, Daniel A Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. The championship simulator: Architectural simulation for education and competition. *arXiv preprint arXiv:2210.14324*, 2022.
- [9] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming*, 2011.
- [10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Waltham, MA, USA, 5th edition, 2012.
- [11] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006.
- [12] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *ACM Transactions on Storage (TOS)*, 2018.
- [13] Magnus Jahre and Lieven Eeckhout. Gdp: Using dataflow properties to accurately estimate interference-free performance at runtime. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
- [14] Harshad Kasture and Daniel Sanchez. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, March 2007.
- [15] Naoki Katoh and Toshihide Ibaraki. Resource allocation problems. *Handbook of Combinatorial Optimization*, 1998.
- [16] ARM Limited. Memory Partitioning and Monitoring (MPAM) for Armv8-A. *ARM Architecture Reference Manual*, 2020.
- [17] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Portland, OR, June 2015.
- [18] Raman Manikantan, Kaushik Rajan, and Ramaswamy Govindarajan. Probabilistic Shared Cache Management (PriSM). *ACM SIGARCH computer architecture news*, 2012.
- [19] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 1970.
- [20] Moinuddin K Qureshi and Yale 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 (MICRO)*, Orlando, Florida, December 2006.
- [21] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Atlanta, Georgia, December 2010.
- [22] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *Proceedings of the 38th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, San Jose, California, USA, June 2011.
- [23] André Seznec. A case for two-way skewed-associative caches. *ACM SIGARCH computer architecture news*, 1993.
- [24] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Waikiki, Hawaii, December 2015.
- [25] Ruisheng Wang and Lizhong Chen. Futility Scaling: High-associativity Cache Partitioning. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Cambridge, UK, December 2014.
- [26] Xiaolin Wang, Yechen Li, Yingwei Luo, Xiameng Hu, Jacob Brock, Chen Ding, and Zhenlin Wang. Optimal footprint symbiosis in shared cache. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.
- [27] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul Chilimbi. All-window Profiling and Composable Models of Cache Sharing. In *Proceedings of the 39th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Portland, OR, USA, June 2012.
- [28] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: A Higher Order Theory of Locality. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, March 2013.
- [29] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.
- [30] Jun Xiao, Yaocheng Xiang, Xiaolin Wang, Yingwei Luo, Andy Pimentel, and Zhenlin Wang. FLORIA: A Fast and Featherlight Approach for Predicting Cache Performance. In *Proceedings of the 37th International Conference on Supercomputing*, 2023.
- [31] Yuejian Xie and Gabriel H Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Austin, TX, USA, June 2009.