CASHT: Contention Analysis in Shared Hierarchies with Thefts

CESAR GOMES, MAZIAR AMIRASKI, and MARK HEMPSTEAD, Department of Electrical and Computer Engineering, Tufts University

Cache management policies should consider workloads' contention behavior when managing a shared cache. Prior art makes estimates about shared cache behavior by adding extra logic or time to isolate per workload cache statistics. These approaches provide per-workload analysis but do not provide a holistic understanding of the utilization and effectiveness of caches under the ever-growing contention that comes standard with scaling cores. We present Contention Analysis in Shared Hierarchies using Thefts, or CASHT,[1] a framework for capturing cache contention information both offline and online. CASHT takes advantage of cache statistics made richer by observing a consequence of cache contention: inter-core evictions, or what we call THEFTS. We use thefts to complement more familiar cache statistics to train a learning model based on Gradient-boosting Trees (GBT) to predict the best ways to partition the last-level cache. GBT achieves 90+% accuracy with trained models as small as 100 B and at least 95% accuracy at 1 kB model size when predicting the best way to partition two workloads. CASHT employs a novel run-time framework for collecting thefts-based metrics despite partition intervention, and enables per-access sampling rather than set sampling that could add overhead but may not capture true workload behavior. Coupling CASHT and GBT for use as a dynamic policy results in a very lightweight and dynamic partitioning scheme that performs within a margin of error of Utility-based Cache Partitioning at a 1/8 the overhead.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Computing methodologies** → **Machine learning algorithms;**

Additional Key Words and Phrases: Last-level cache, cache partitioning, gradient-boosting trees

ACM Reference format:
Cesar Gomes, Maziar Amiraski, and Mark Hempstead. 2022. CASHT: Contention Analysis in Shared Hierarchies with Thefts. *ACM Trans. Archit. Code Optim.* 19, 1, Article 12 (January 2022), 27 pages.
https://doi.org/10.1145/3494538

1 INTRODUCTION AND MOTIVATION

The number of cores on a chip continues to increase, which adds pressure to scarce and shared resources like the **last-level cache (LLC)** [26]. Though shared resources are constrained by area and power, there is increased demmand for more computing power [4]. Workloads are also growing in complexity [40], and virtualization obscures underlying hardware, creating dissonance between

[1]New article, not an extension of a conference paper.

Authors' address: C. Gomes, M. Amiraski, and M. Hempstead, Department of Electrical and Computer Engineering, Tufts University, P.O. Box 1212, Medford, Massachusetts, USA; emails: cesar.gomes@tufts.edu, {maziar, mark}@ece.tufts.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1544-3566/2022/01-ART12 $15.00
https://doi.org/10.1145/3494538

promised and available compute resources. **Service-level Agreements (SLA)** mitigate dissonance by promising a quantifiable expected performance as **Quality of Service (QoS)** to users [2, 9, 41]. However, servers are increasingly highly utilized with programmers and administrators squeezing as much performance and throughput from available hardware as possible. Growing demand for computing means resource contention is a persistent and dominant characteristic of many-core designs barring paradigm shifts in computer architecture.

There is an information gap between stochastic cache behavior that misses cannot fill at scale. Misses are a foundational measure of the utility and efficacy of hardware caches. A miss occurs when a request for a block of data is not fulfilled by the cache, and the types of misses that occur vary: inevitable, or compulsory; a consequence of no space, or capacity; a consequence of too much current and relevant data, or conflict; or a matter of data sharing in the case of SMT, or coherence. However, the first three of the traditional classes of misses are classified assuming only a single workload is executing. Traditional misses are often indirectly attributed to contention when run with other workloads, but accounting for that requires additional tracing of which blocks are impacted by such events in addition to the type of miss. A simpler approach is to study the consequences of cache sharing directly, which includes cache evictions. We investigate cache eviction, identifying the cache occupants involved in the cause (eviction) and effect (evicted) to determine if the eviction is due to accesses from the workload that inserts the block being evicted or from another workload. Please note that coherence misses are often considered in multi-threaded contexts, and though there are instances where different programs exploit data sharing, we will not be investigating coherence misses in this work.

Scaling to inform cache partitioning is costly for hardware solutions. When partitioning a cache one can consider numerous solutions from the literature including a well-known re-partitioning algorithm: **Utility-based Cache Partitioning (UCP)** [32]. UCP (and notably the LookAhead Algorithm) is often evaluated in related work or used as foundation for newer algorithms [11, 35, 45, 48]. Consequently, UCP assumes that hit curves that are inputs to the LookAhead algorithm are approximating single workload behavior by using separate sampling structures for each workload. The downside of such an approach is that we must add sampling structures per core as we scale core counts higher. We avoid this cost by collecting cache statistics in a probabilistic way, taking advantage of the trained learning model that learns and predicts the best ways to partition cache and an algorithm that scales this solution to >2 cores.

*We present **Contention Analysis in Shared Hierarchies with Thefts (CASHT)**.* CASHT provides a framework for the development of lightweight and contention-aware re-partitioning algorithms that compare well against UCP. Prior art often measures contention indirectly through variations in misses or IPC, or directly through events that signify a difference between solo and shared cache occupancy [11, 25]. CASHT utilizes **THEFTS**, a measure of cache contention in the form of intra-occupant cache evictions that encode a cache eviction with the context of the interaction between cache occupants. However, informing partitioning algorithms with **THEFTS** is difficult while partitioning. We present **Agnostic Contention Estimation (ACE)**, which detects when partitions *prevent* thefts with reduced overhead (0–0.2% of the cache). Tracking contention in caches with more and more cache occupants offers useful, contextual, and complementary information in designing better and/or lighter re-partitioning logic. However, developing a re-partitioning algorithm from scratch requires time, resources, and a deep understanding of the relationships in the data set. CASHT takes advantage of machine learning models to save for all three of these critical dependencies. We train **Gradient-boosting-tree (GBT)**-based supervised learning models with feature sets containing cache statistics collected in the context of contention, taking

advantage of the complementary nature of contention to inform cache needs and avoid the need for extra time or logic overhead to mitigate influence over data collection [8]. GBT-based models offer the opportunity to borrow directly from the conditional tree generated by GBT model to create lightweight logic for use in partition prediction (1–10 kB). We show that contention unaware models are at a disadvantage in comparison to contention-aware models. The contributions of this work are as follows:

- The insight that contention-unaware data does not offer the best partitioning solutions (best, fair, QoS);
- Thefts and Interference: a direct measure of cache contention through inter-core evictions;
- ACE: method of capturing theft-based contention despite partitioning;
- PSA: sampling framework built on ACE that allows per access rather than subset sampling, frees cache from added sampling overheads, and enables full cache partitioning (no subset of non-managed cache sets);
- GBT-based Re-partitioning: a lightweight, high accuracy learning model trained on contention-aware data set;
- CASHT: GBT-based re-partitioning framework that employs probabilistic sampling, agnostic contention estimation, and a novel algorithm for scaling a GBT model trained on two-core data to workloads of more than two cores.

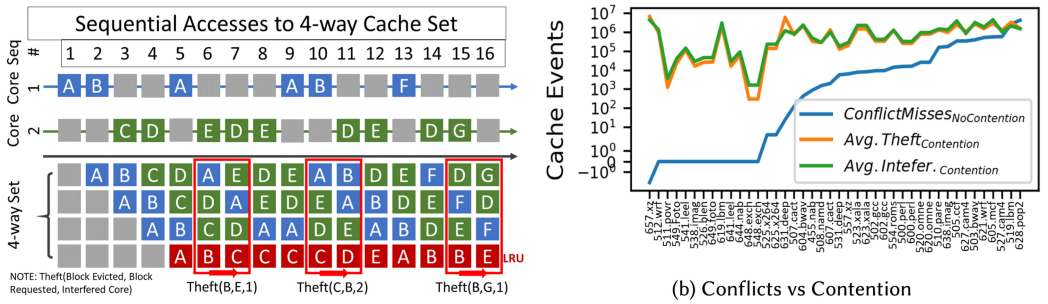
2 THEFTS—MEASURING CONTENTION

Contention is a matter of course in multi-cores and capturing contention provides insight into how workloads share or fail to share resources. Taking the shared nature of resources into account offers architects and designers an opportunity to measure contention, utility, and performance simultaneously. We present a new measure of contention called **THEFTS** that correlates miss events with interference in shared caches by counting inter-core eviction. We collect theft-based stats, misses and IPC data shown in this section from 860 two-core simulations and 42 single-core simulations in the environment that we detail in Section 6. The simulations assume an unpartitioned last-level cache.

2.1 Thefts—Evictions Not Induced by Inserting Workload

We define **THEFTS** as workload interactions in the last-level cache that result in an eviction. Counting thefts requires capturing these interactions, which we show in Figure 1(a). Given unique data request streams from two cores, we see how both share a four-way cache employing the least recently used (LRU) as the eviction policy to choose which block is removed first. The first **THEFT** happens at sequence #6 where core 2 requests data block E, cannot find it in the cache, and needs to evict the LRU block (B) from the set so E can be written. Core 1 inserted Block B at #2, so core 2 evicting block B means core 2 executes a **THEFT** of resources from core 1. We see similar occurrences at #10 (Core 1 executes theft on Core 2) and #15 (Core 2 executes theft on Core 1). In this way, we can capture a new event with a simple equivalence comparator that is out of the critical path. Further, the context of interaction means there is also perspective, i.e., if we look at the first theft event, then we see that core 2 executes a **THEFT** and core 1 experiences **INTERFERENCE**. Moving forward, we refer to the execution and experience of thefts as **THEFTS** and **INTERFERENCE**, respectively.

Thefts can result in misses but not all misses are thefts. Given that thefts are a type of eviction, we must consider the relationship between misses caused by evictions, or conflict misses. Figure 1(b) compares conflict misses captured in isolation to cache thefts and interference. The order of magnitude difference between conflict misses when compared to thefts and interference



(a) Capturing Thefts

(b) Conflicts vs Contention

Fig. 1. Theft Example and Comparison to Conflicts: Thefts, or inter-core cache evictions, differ from miss-based metrics like conflict misses, because the metric attributes applications/cores with being the cause rather than architectural limitations. In panel (a), we see three thefts occur across 16 consecutive accesses to a four-way cache that two cores share, with one insertion by core 1 (access 10) and two insertions by core 2 (access 6 and 15) causing thefts. In panel (b), the x-axis lists each workload and the y-axis shows the log scale of cache events; we compare misses saved by doubling associativity, or conflict misses captured when workload is run alone, against average thefts and the related interference for each workload when they share cache with another workload. The comparative difference in conflicts and the contention metrics shows the removal of conflicts is not the same as the prevention of cache contention and thus differentiates thefts from conflicts.

show that contention is not a consequence of a workload not fitting in cache but of forcing applications to contend in a limited space. Certainly, we can consider the working set of the mix as one working set and take thefts as conflict misses, but we lose the unique and nuanced behavior of each workload.

2.2 How Do We Measure This? (NCT Algorithm and Overhead)

Detecting thefts is a simple process that requires simple modifications of miss detection logic: adding a core or thread ID comparator, and access type comparator. We assume the system represented in this and the remaining algorithms do not employ **simultaneous multi-threading (SMT)**, so the CPU ID indicates physical core ID. In the case of SMT, a thread ID could be used. Algorithm 1 describes how we employ **native contention tracking (NCT)** to detect when evictions of valid cache blocks are thefts. On a miss, we check whether the cache block chosen by the eviction policy is valid and whether the CPU ID of the block and the CPU ID of the accessing CPU are different. If this holds, then we have detected contention and can update counters. Assuming the access type is not a writeback, we can update the theft counter for the accessing CPU and the interference counter of the CPU that initially inserted the eviction candidate block. If the access type IS a writeback, then we DO NOT update the theft counter. The reason for not updating the thefts counter is because we want to make thefts a distinct action taken by the related CPU, not a consequence of the upper-level caches not having the capacity to hold modified data.

2.3 Statistical Analysis (Pearson, Spearman) Versus Misses

Cache statistics are often used in characterization studies and feature heavily in results coming from simulation environments [7, 19, 22, 34]. Commonly used statistics include cache hits, misses, evictions, and these can be broken down further by access type. Such metrics still contribute great information for analysis, but growing cache hierarchies hide and obscure relationships that

ALGORITHM 1: Native Contention Tracking (NCT)

```
Result: Updates Contention Counters
A = associativity;
eB = cache[S][W];
C = CPU ID;
S = set index;
T = access type;
W = way index;
hit = cache hit boolean;
if not hit and eB.valid and not(eB.cpu == C) then
    if T!=WRITEBACK then
        Thefts[C]++;
    end if
    Interference[eB.cpu]++;
end if
```

statistics like hits and misses are frequently used to determine. For example, we have reached a point with deep cache hierarchies and scaling cores that misses can mean something dire or simply be a consequence of the application. We believe misses and other familiar information lack context of the new shared cache paradigm and should be offset with contention information like thefts. To demonstrate this, we show results from conducting Pearson and Spearman statistical significance tests on miss-based heuristics like Miss Rate ($\frac{misses}{accesses}$), **Misses per 1,000 Instruction (MPKI)**, and similarly formulated theft- and interference-based heuristics in Table 1. Each cache statistic data set is tested against a data-set composed of the **instructions per cycle (IPC)** from a common set of 860, different two-workload trace experiments. All features are normalized between 0 and 1 per respective features (for example, all thefts are normalized between 0 and 1 according to the maximum and minimum theft across all experiments). Pearson tests determine the linear correlation, or whether two data sets are linearly independent by computing a correlation coefficient (R) bound between −1 and 1 (0 means little to no correlation) and the P-value, which indicates if the result is statistically significant ($P < 0.05$ or 0.1 often acceptable) [6]. Spearman rank correlation determines if two sets of data can be described by a monotonic function, and has similar implications regarding R- and P-values. While not as strong in all cases, theft- and interference-based metrics have a clear statistical significance (P well below 0.05). In fact, the correlation of thefts per miss demonstrates that thefts complement and are complemented by misses, and they help characterize potential relationships between misses due to contention and performance.

3 MEASURE THEFTS WHILE PARTITIONING IS IN PLACE

Our analysis shows thefts and theft-based metrics are correlated to performance, and are comparable and complementary to misses in the Last-level Cache. However, allowing such contention is not a favorable choice for designers eager to mitigate it. Cache partitioning, insertion, promotion, and other policies target contention mitigation either directly through physical separation [15, 23] or indirectly through predicting when to leave blocks vulnerable to eviction or bypassing cache altogether [20, 21, 29, 31, 46]. Getting a true measurement for theft-like contention is nearly impossible while such mitigation methods are in place, but we have an estimation framework that can estimate contention despite techniques that prevent it. We discuss a lightweight method

Table 1. Comparison of Correlation and Statistical Significance

Metric vs. IPC	Pearson R	Pearson P	Spearman R	Spearman P
MPKI	−0.29	3.94e-27	−0.37	1.05e-42
$\frac{Misses}{Accesses}$	−0.09	0.0	0.03	0.26
TPKI	−0.19	2.87	−0.23	5.04e-17
$\frac{Thefts}{Misses}$	0.48	2.09e-76	−0.24	2.50e-18
IPKI	−0.20	1.57e-12	−0.24	6.08e-18
$\frac{Interf.}{Misses}$	0.10	0.0	0.23	8.74e-17

ALGORITHM 2: PSA + ACE

```
   Result: Detects Prevented Theft with some Probability
1  A = associativity;
2  S = set index;
3  W = way index;
4  C = CPU ID;
5  T = access type;
6  B = cache[S][W];
7  hit = cache hit boolean;
8  pT = SampleSetCount/LLCSetCount;
9  if RandomNumberGenerator/RAND_MAX < pT then
10     if not hit and not(B.lru==A-1) and B.valid then
11         if T!=WRITEBACK then
12             theft[C]++;
13         end if
14         trueLRU = get_LRU(cache[S]);
15         interference[cache[S][trueLRU].cpu]++;
16     end if
17 end if
```

for collecting and sampling cache contention. First, we present ACE, a framework for estimating so-called "prevented thefts" in a cache that may have partitioning or other cache management policies in place. ACE takes advantage of the LRU stack to count thefts **and** interference on cache evictions that result in non-LRU blocks being evicted from LLC. ACE has the nice benefit of avoiding additional per block CPU IDs, which can be costly at scale. Further, having the ability to count contention regardless of the cache mitigation method in place affords us an opportunity: sampling on a per access basis. Sampler logic in recent work assigns specific sets to be sampled from, but leaves open the possibility that not all sets are accessed or provide information. We demonstrate a probabilistic sampling method, Probabilistic-ally Sampled Accesses or PSA, which takes advantage of ACE to sample on any given access with some probability.

3.1 Agnostic Contention Estimation (ACE) Algorithm

Agnostic Contention Estimation, or ACE affords us the ability to track contention agnostic of the contention-mitigation methods enforced in the cache. Specific to cache partitioning, ACE leverages the LRU stack to determine when a partition prevents eviction of the true LRU when that block is in another partition. ACE tests if the current eviction candidate provided by the replacement

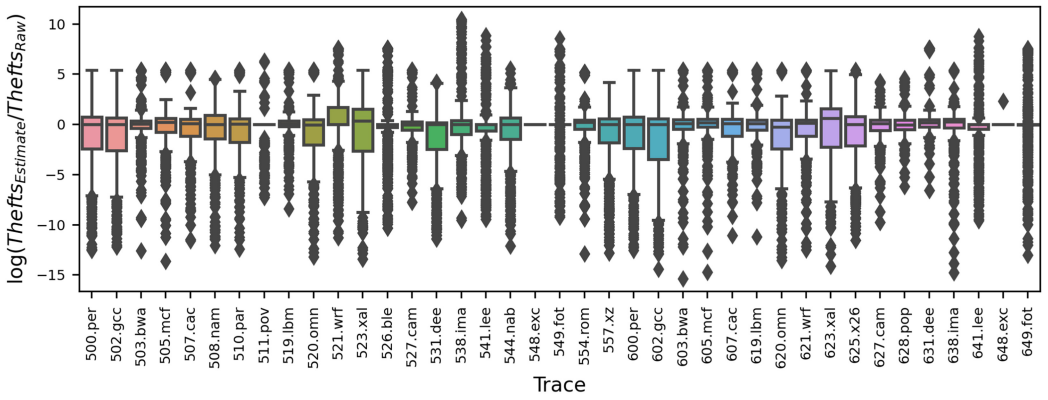


Fig. 2. Comparing True and Estimated Thefts: We normalize the estimated thefts from our cache-partitioning sweeps by the workload-respective true thefts collected under contention, with the y-axis representing the log of this normalization; We see that theft estimates vary by orders of magnitude, which can be contributed to the different partitioning allocations we collect theft estimates from biasing cache availability and therefore increasing the likelihood of estimated thefts. It is clear that we cover real theft behavior due to most averages being near zero, but the wide ranges indicate sensitivity to partition configuration.

policy is LRU on a cache miss. If the candidate is not LRU, then we traverse the set until we find either the LRU block or the block with the highest LRU value exclusive of the eviction candidate. To avoid double-counting of prevented contention, we skip blocks that have the theft bit set, which indicates we prevented an eviction on this block on previous access. If we find a block that meets our criteria, then theft estimates for the CPU inserting a new block and interference estimates for the CPU that inserted the protected block are incremented **only** if the CPU identifiers do not match. ACE does have time and area overhead when employed as we describe (Section 3.3), but we simplify this by comparing the LRU stack position of a replacement candidate to LRU to avoid CPU ID overhead. The time to traverse the set for the next nearest replacement candidate can also be avoided by finding the set-wide LRU and using the associated CPU ID to update interference counters.

Figure 2 compares our theft estimates from ACE to thefts captured in an un-partitioned cache. We do this by normalizing theft estimates collected in all possible partitioning configurations to real thefts captured in the un-partitioned cache, which we illustrate on the y-axis (higher than 1 means over-estimation, and lower than 1 means under-estimation). The x-axis shows the name of the benchmark our traces are derived from, and data is represented as box plots, because for each trace we have 15 times 41 different data points from our partitioning studies. The figure shows the mean of most box plots is near 1, so our estimates do capture expected behavior, but the upper and lower bounds are orders of magnitude away from this mean. Such wide ranges of normalized theft estimates speak to the capacity sensitivity and utility of partitioning solutions. Further, estimating contention with ACE has the consequence of enabling per access sampling. Cache set sampling is the common method of collecting cache statistics on certain cache sets that the architect designates at design time, either as an **Associative Tag Directory (ATD)**, which needs additional hardware, or **In-cache Estimation (ICE)**, which needs a subset of cache not managed like the rest of cache [32, 48]. Prior work employs these techniques to great effect [21, 31, 32, 46], but only sample accesses to selected sets, which runs the risk of misrepresenting workload behavior and can lead to different conclusions about a given workload.

Table 2. Sampler Details and Average Sampled Hit Rate (HR)

Sampler	Sample accesses	Overhead	HR_{Sample}/HR_{Full}
ICE	To a subset of un-managed cache sets	None	102
ATD	To a separate associative structure	N*M*(tag+LRU+valid bit)	102.98
PSA	With some probability, P	None	99

3.2 Probabilistically Sampling Accesses (PSA) Algorithm

Modern cache sampling logic is built such that the number of cache sets chosen to be sampled implies a ceiling on cache accesses to be sampled. We can define this ceiling as follows:

$$\lceil P(Sample) \rceil = \frac{s}{S}, \tag{1}$$

where s is the number of cache sets designated for sampling and S is the total number of cache sets. The concern with the sampled access ceiling is that the amount of sampled accesses may never approach it, because not every designated set (fixed or randomly selected) may be accessed by any workload. PSA employs the sampled ceiling as a probability threshold over which no statistical accounting can occur.

A comparison of sampler hit rates in Figure 3 shows PSA replicates full workload hit rate more reliably than both ATD and ICE. The summary in Table 2 shows PSA captures 99% of the full hit rate for SPEC 2017 traces on average while ATD and ICE over-estimate (are optimistic) hit rate by 2.98% and 2%, respectively. The data in Figure 3 shows sampled hit rates captured by each of the sampling techniques normalized to full hit rate. ATD and ICE are configured such that candidate sets are equidistant from each other across the cache. Because PSA collects lines with some probability, 25 simulation iterations are taken per workload and represented as a box plot. Please note that we use the C/C++ random library, and seed it with the time at the start of each simulation.

Workloads that ATD and ICE over-estimate are captured fairly accurately by PSA with (619.lbm, 511.povray, 641.leela, 541.leela). 538.imagick indicates a lower bound on the range of hit rates seen across PSA iterations that are far lower than what ATD and ICE represent, and an additional set of workloads (511.povray, 648.imagick, and 603.bwaves) indicate PSA simultaneously over- and under-estimates hit rate. The behavior can be attributed to workloads having multiple working sets with different hit rate behaviors being captured by PSA run with a different time seed. Such behavior indicates PSA sensitivity to different behaviors across a workload that lends well to prefetcher training or other dynamic policies hoping to capture distinct behavior, though further investigation will have to wait for future work.

3.3 Algorithm Description and Overhead

Algorithm 2 shows how PSA and ACE come together. The algorithm has a similar structure to Algorithm 1 except now statistics collection happens depending on the result of the random number generator from PSA at line 9. Also, now we use ACE at line 10 to detect if the replacement candidate is the set-wide LRU (comparison to the max LRU value, associativity-1), and again at line 14 to find the set-wide LRU block to update interference. ACE requires a bit to be added per block to enable correct theft and interference accounting, which translates to 8 kB for a 4 MB LLC and scales with cache size. PSA requires logic for a random number generator and comparator logic for the current probability and the sampling threshold we impose. Hardware random number generators can come with a cost, but recent efforts see low power, low area, accurate RNGs that can be included in our design [24, 30, 50]. Finally, since we are sampling contention on any given miss with some probability due to PSA, we can modify line 10 to test if an eviction candidate

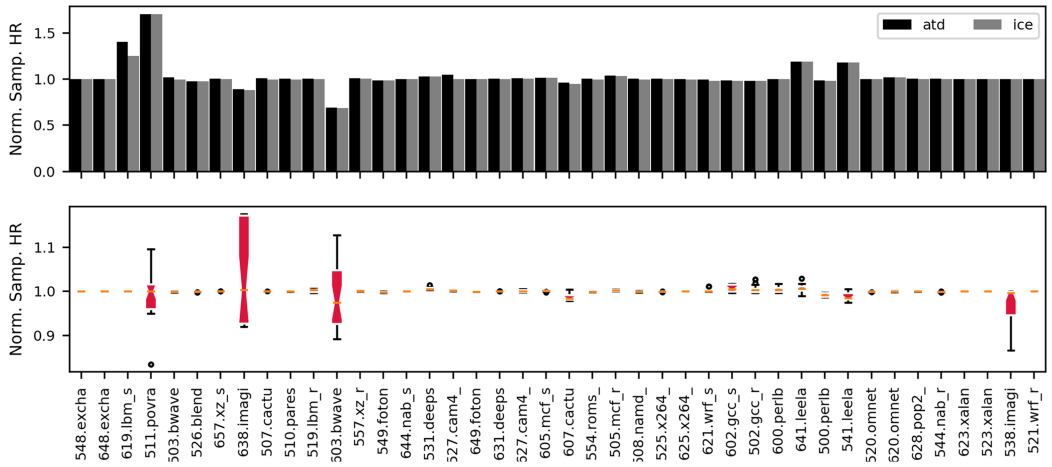


Fig. 3. Sampler Comparison: We compare hardware sampling techniques to determine how well they capture hit rate, normalizing the sampled hit rate each sampler collects (ATD, ICE, PSA) to the full hit rate collected from a simulation of LRU. ATD and ICE are configured such that the sampler sets are at equal distances away from each other across cache while PSA samples per access with some probability, so we run 25 iterations of PSA to cover a range of possible outcomes. We observe that while ATD and ICE have some workloads where hit rate is over- or under-estimated, the mean value that PSA reports is often 1, or similar to full hit rate. The instances of wide standard deviation indicates diversity in working sets across those workloads that are not captured in a fixed set solution.

is LRU, remove line 14, and avoid the cost of an additional bit per block. For comparison, UCP requires 3.7 kB per core for each Associative Tag Structure, which scales with core count, while PSA sees no additional memory overhead aside from the hardware counters for thefts and interference per core.

4 SUPERVISED LEARNING ALGORITHM

Machine learning has recently shown promise when applied to system problems [5, 10, 14, 39]. However, the challenge is providing implementations that are lightweight both in the structure of the predictor, and the feature extraction cost during system run-time. In this section, we explore the use of a machine learning model, and the concept of thefts, to choose the best partitioning configuration based on features extracted from each core and every level of cache.These features are: (1) Access, hit, miss, miss-rate, and MPKI of different levels of cache hierarchy, namely, L1D, L1I, L2, and L3, (2) IPC, and (3) Thefts, Theft-rate, and TPKI from LLC.

4.1 Choosing a Learning Model

Similar to Ren et al. [33], we explore different learning to determine a good fit to the prediction problem, and start with **Multi-level Perceptrons (MLPs)**. However, these fully connected models are pretty expensive due to their high number of weight parameters. We next test pure Decision Trees but achieve very low accuracy, and then Random Forests, which improves the accuracy to some extent. We have also explored employing logistic regression and SVM models for our training purposes. Comparing the accuracy of all these models, we decided to use Gradient-boosting Trees, or GBTs [12]. Decision trees, at the core of GBTs, have the following satisfactory properties that make them beneficial for our study: These models do not require pre-processing such as

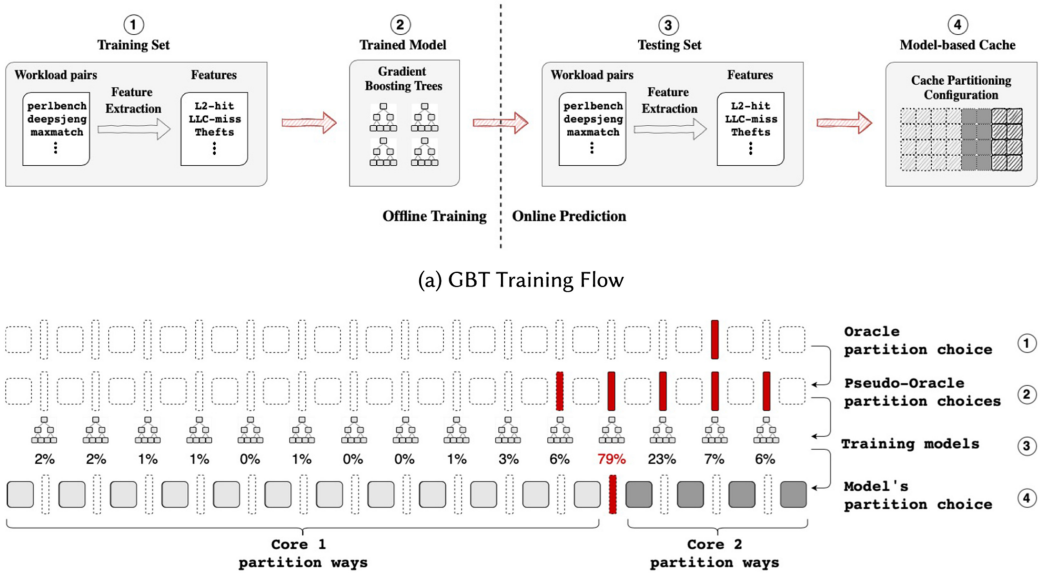


Fig. 4. (a) First ①, we create workload mixes and extract features from them. This dataset is then divided into a training and a test set. Second ②, using the training data-set, we train GBT models to predict the best partitioning configuration offline. Next ③, we extract features from the test set. ④ We use the trained models to infer the partitioning configuration during the system's run-time operation. (b) Multi-label training for gradient-boosting trees to predict partitioning configuration. Squares show cache ways and bars show where we can partition the cache (inspired by stars and bars in combinatorics). The first row ① shows the Oracle partition configuration. The second row ② shows possible pseudo-Oracle partition configurations (where the IPC of the resulting system would be at most 1% less compared to oracle's IPC). The third row ③ shows the models trained on each partition choice (bars) and their output confidence. The last row ④ shows the partition configuration chosen by the model. The model chooses the configuration with the highest confidence found on the third row, dividing the cache between two cores.

feature normalization on data; we can easily visualize and analyze them; and implementing them in hardware is easy due to the simplicity of tree logic. In addition, as we will describe shortly, they can solve *multi-label problems*. One major disadvantage of decision trees is that they could easily over-fit and have lower prediction accuracy compared to other more complicated models. We discuss how to improve decision tree results and to prevent over-fitting with ensemble techniques in this section.

4.2 Gradient-boosting Trees

We use the gradient-boosting method for this study. In gradient boosting, several shallow trees are trained and connected in a serial manner, where the residuals of each tree are fed as input to the next tree. This way each subsequent tree would gradually improve predictions of the previous tree. The simple structure of decision trees combined with gradient boosting is the sweet spot we were looking for to decide on a partitioning configuration. Figure 4(a) shows a high-level flow for how we train a GBT model. To reach an acceptable accuracy, we train our model on more than 600 different mixes of application pairs. After creating the workload combinations and extracting their

features through simulation, we trained a model offline on the feature sets. The trained model is then used on unseen mixes to predict a partition, approximating the oracle configuration.

4.3 Defining the Multi-label Prediction

We devised the partition prediction problem as a multi-class multi-label problem, as shown by Figure 4(b). The rationale is as follows: the overall goal of our model is to choose a partitioning configuration to achieve the best IPC. This IPC is the result of the Oracle partitioning configuration that is shown in the first row in Figure 4(b). The label for this example is (000000000000010). The index of 1 shows where we have to partition the cache to achieve the optimal IPC. However, by observing our dataset, we learned that some of the Oracle configuration's neighbours have an IPC that is pretty close to the Oracle. This inspired us to put a threshold in place where if the IPC of another partition choice is within 1% of optimal IPC, then it will also be counted as an acceptable configuration. These pseudo-Oracle partition configurations are shown in the second row. The label for this example is (000000000001111). Therefore, for each application pair, we can have one to several partition choices and this will make it a multi-label problem.

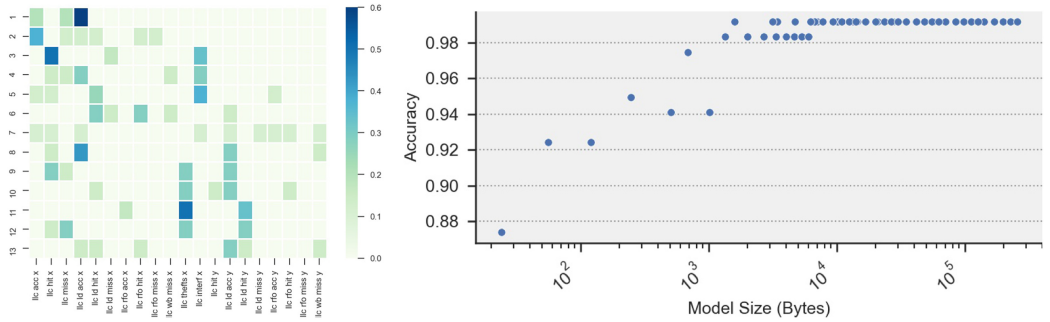
Suppose that we have N ways and we want to divide it between two cores. A possible configuration is to give 1 way to application one and N − 1 ways to application two. Or 2 ways to application one and N − 2 ways to application two. Increasing the number of ways given to the first application would decrease the number of ways given to the second application and vice versa. The goal is to train a model that, based on features extracted from each core, would tell us where to partition the cache to achieve the highest IPC for the system. These models are shown in the third row of Figure 4. We can see from the figure that for a cache that has N ways, there are N − 1 locations that we can partition the cache between two cores (shown by bars in Figure 4(b)) considering that each core gets at least one way. We will then train a GBT for each of the N − 1 partition choices.

4.4 Training GBT

The training is done offline on all the instances of the training set. To train the models using our supervised learning algorithm, we need input feature vectors and true labels for each instance. We have collected the input features through extensive simulations. Additionally, the label for each GBT would be either 0, meaning that we should not partition in that specific location, or 1 meaning that we should. Using the features and labels we train our models and the result of this stage is a group of trained GBT models. Next, we use instances in our test set to receive a prediction on where to divide the cache between cores. The third row of Figure 4(b) shows the outcome of doing a prediction using GBTs on one of our test set instances. This result comes in the form of the model's confidence on where the optimal position for the partitioning should be. We will choose the partition configuration with the highest confidence and assign cache ways to cores based on that prediction (fourth row).

4.5 False Predictions

Taking into account our choice of problem definition, it is apparent that false positives have much more importance compared to false negatives. False positives show configurations predicted by the model to have optimal IPC while they do not, and false negatives are ones that models predicted not to have optimal IPC while they do. We are not concerned about false negatives as long as our model produces at least one true positive result. This positive result should be either the optimal partition choice or one of the other partitions (if any) that has an IPC difference of less than 1% from optimal IPC. However, false positives should be avoided, since they could penalize system performance.



(a) Feature importance heatmap (b) Cost vs. accuracy

Fig. 5. (a) Feature importance in a low-cost high-accuracy model. The x-axis shows the features, and the y-axis shows the location of the partitioning. The darker the color, the higher the importance of the feature in partitioning prediction. As we can see, Thefts and interference have an important role in choosing an optimal partitioning configuration; (b) Accuracy vs. cost. The x-axis shows the size of models in Bytes. The more and deeper the trees, the larger the size of the models. The y-axis shows the accuracy of the models. This is the ratio of the correct predictions by the models, to the correct labels. Generally, the larger the tree, the higher the accuracy. However, this is not always the case.

4.6 Feature Importance Study

One question that remains is how important are the specific features introduced in this work, namely, Thefts and Interference, in describing our models. To answer this question, we first conducted a study on highest accuracy achieved by training models using either features from all levels of cache or LLC only. The accuracy of these models were pretty close. However, it was more reasonable to use LLC-only features due to higher cost of passing and maintaining the core cache data when the accuracy is the same. Using LLC statistics, we explored the feature importance for one of the low-cost high-accuracy models. The result is shown in Figure 5(a). The x-axis shows the top 20 important features in the model, and the y-axis shows the location of the partitioning configuration. The darker the cell in this figure, the more important the feature to select an optimal partitioning located in that location. As we can see in this figure, the importance of Thefts and Interference is more pronounced in the middle locations compared to the extremities. This was expected, since there is considerably more contention between workload pairs that mutually require larger partitions, compared to the pairs where at least one application needs a small partition.

4.7 Overhead

We discuss selection and training of a supervised learning model, gradient-boosting trees, on a two-core mix data-set and employ it to predict the last-level cache-partitioning configuration with the highest system IPC. We use features extracted from last-level cache, which include thefts, MPKI, and so on. We define the partitioning problem as a multi-label problem and produce several, correct labels per two-trace pair. To achieve an acceptable accuracy using our models, we need to tune their many hyper-parameters. Utilizing XGBoost library [8], these hyper-parameters include the number of trees, the maximum depth of trees, the learning rate, the sampling ratio of training instances, and so on. We grid-searched these hyper-parameters and did fivefold cross-validation on the training set to attain a good degree of confidence in the accuracy of our models. Figure 5(b) shows the cost versus accuracy plot of these models. The x-axis in this plot shows the size of the model in Bytes and the y-axis shows the accuracy of the model, predicting one of the correct partitioning configurations. As we can see, the smaller models have lower accuracy, but it is not

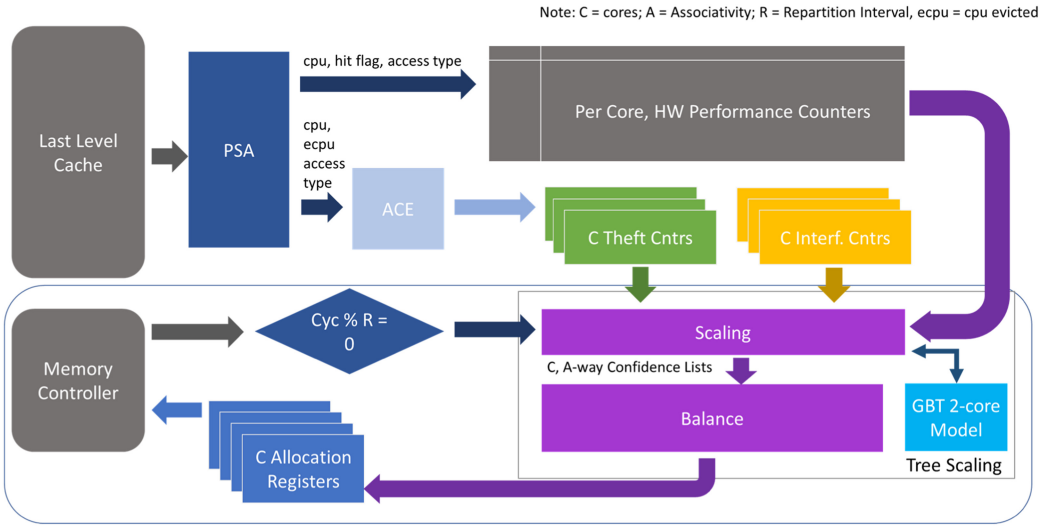


Fig. 6. Contention Analysis in a Shared Hierarchy with Thefts: We show the high-level format for CASHT in this figure, demonstrating how it can be integrated into a common cache hierarchy. We integrate PSA at the last-level cache (LLC) to probabilistic-ally update LLC-level HW performance counters and leverage ACE to detect when thefts/interference occur. At the memory controller, we re-partition every R cycles, leveraging GBT through our tree-scaling algorithm, which enabled our two-core model to be useful at a higher core count.

necessary to have the largest models to achieve the highest accuracy. On the contrary, some of the largest models show lower accuracy compared to smaller models. This is due to the models over-fitting when the trees get too deep or too many.

5 CASHT: CONTENTION ANALYSIS IN A SHARED HIERARCHY WITH THEFTS

We present Contention Analysis in a Shared Hierarchy with Thefts, or CASHT, a framework that takes advantage of minimal contention estimation (ACE), sampling logic (PSA), and a partitioning prediction engine (GBT) to generate re-partitioning solutions at run-time with little overhead. Figure 6 illustrates how ACE, PSA, and GBT come together to create CASHT. We integrate PSA and ACE at the LLC, and PSA determines when LLC-related hardware counters are updated. Critically, ACE checks if an eviction is a theft (and subsequently causes interference) only PSA allows sampling on that particular access. We integrate GBT at the memory controller via an algorithm that takes advantage of the pseudo-oracle prediction list, or **configuration confidence (CC)** list that GBT outputs to scale our model to higher core counts. We call this algorithm Tree Scaling and use this to determine the next partition allocation for two to eight cores in our experiments.

5.1 Tree Scaling: Algorithm that Scales our Two-core Solution to Four+ Cores

The high accuracy of the GBT model at two cores motivates interest in a model that can predict for higher core counts, but the effort to generate the data to do so is prohibitive. For example, to find the best configuration for a four-core mix, we would need to simulate 455 different simulations! We present Tree Scaling, an algorithmic approach to enable a GBT model, which trains on features from two-core simulation results to be of use at higher core counts (4+ cores). Tree scaling takes advantage of the multi-label confidence output or configuration confidence (CC) list that GBT

generates to reason about how to distribute cache allocations on >2 core systems. Tree scaling has three hyper-parameters (T, D, and s_{max}) and two components: Scaling and Balancing.

5.1.1 Hyperparameters. We design tree scaling with four hyper-parameters to control how allocations are distributed: a confidence threshold, T; a threshold decay rate, D; and a provisioning switch event maximum, s_{max}. The confidence threshold indicates the confidence level that a configuration in a CC must meet to be selected as a new partition. The threshold decay rate is the amount we decrement the current threshold in the event we cannot find a solution or we have switched provisioning schemes too often and might have missed a solution. We track how often the total allocation becomes successively over- and under-provisioned without finding a balanced solution. We compare the number of times this occurs to a switching threshold, or the number of times tree scaling can switch provisioning schemes before subtracting the decay value from the current threshold. For brevity, we have excluded the analysis from this work, but we find the best performance with 0.1 and 4 for the threshold decay and switch count max, respectively. We set s_{max} equal to the number of cores. Design space explorations will be done in future work.

5.1.2 Scaling. Tree Scaling generates CC lists per core by placing each workload as the first input feature set and a combination of features from other workloads as the second input feature set. For example, say we want to generate a CC for core 0 in a four-core mix. Recall that GBT takes N features per core (total features = N*core) to predict confidences for each way of dividing cache between 2 workloads and represents this as a (set associativity-1)-element list of confidences bound between 0 and 1 (what we refer to as a CC). Tree scaling takes two steps toward generating the CC for core 0 by creating an (N*2)-element input list to GBT, assigns the N features for core 0 as the first N of the input list, and does an element-wise combination of the N features for all remaining core features. For example, if we have hits, misses, and thefts for each core, then the input list looks like the following: $\boxed{hits_0}\boxed{misses_0}\boxed{thefts_0}\boxed{hits_{1:3}}\boxed{misses_{1:3}}\boxed{thefts_{1:3}}$. We combine non-theft features with a sum while rates and theft-based features via a max function. Taking care to combine thefts differently from other features is necessary given that thefts and interference are a consequence of sharing last-level cache, and are therefore dependent on the other workloads that share last-level cache. Once complete, there will be a CC per each core that shares cache, and we can traverse to find the allocation with maximum confidence at the smallest configuration (MaxMin). The resulting output is a list of partition solutions for each core that we pass to the Balancing component.

5.1.3 Balancing. When the sum of the output from *Scaling* does not equal the associativity of the cache, we must resolve this over- or under-provisioning of resources. Tree scaling handles miss-provisioning in two ways: if under-provisioning, then the partition with the most to gain from increasing the current allocation is selected (calculate the average weight of allocations greater than current allocation); and if over-provisioning, then the partition with the least to lose from decreasing cache resources is chosen (calculate the average weight of allocations lesser than the current allocation). We calculate most-to-gain by selecting all of the configurations with highest average confidence greater than the current selected configuration. For example, in a two-core system, we compute the max like this: $max(\sum_{j=maxmin[i]}^{A} CC[i][j], i \in [0, \ldots, C])$, where A is associativity and C is core count. Similarly, we calculate least-to-lose, except we do this computation for all configurations less than the current configuration.

Avoiding Infinite Loops. We address two cases where Tree Scaling can loop infinitely by decaying a probability threshold: if we cannot find a solution; or if we switch between over- and under-provisioned when searching for a solution. In the event of either case, we decrement T by the

threshold decay value, D. Our strategy takes inspiration from hill-climbing algorithms that revise criteria when an answer is not found [16]. We choose decay sizes in accordance with the severity of the problem: provisioning state toggling reflects the algorithm circling some suitable (and fitting) solution so small steps are appropriate (T-0.01); decrementing T in small increments numerous times suggests the need for a more drastic reduction (T-0.05). An example for causing large decays is if the algorithm is in a toggle state and achieves the first decay condition, then enters the over-provisioned state, and then reverts back to the toggle state.

Optimizing when Equivalent. There are conditions where solutions and even partition allocations are equal that require addressing. While balancing, the corner case where all cores have equal to gain or lose if the allocation changes can lead to an infinite loop and is avoided by comparing the average confidence of the whole CC for each core: we increment the partition with the smallest average confidence, choosing a configuration above a confidence threshold, T; or decrement the partition with the largest average confidence, choosing a configuration above a confidence threshold, T. Additionally, we enforce a **Fair** distribution of capacity when the minimum best configuration as designated by all CCs is the smallest and all solutions per each CC have equal weight (for example, all weights in the CC == 0.90).

The exit condition for the tree scaling algorithm is when the sum of the allocations is equal to associativity, we set the new allocations. Consequently, the newly generated list of partition solutions to enforce for the next 5M cycles is what returns to the memory controller. The tree scaling algorithm allows CASHT to scale to more than two cores, and we show the performance results for four and eight cores in Section 7.

5.2 Tuning

We tune CASHT by sweeping the contention collection method, the sampling rate or probability of sampling on a given access, and the rate that we re-partition cache. The performance metrics we analyze are average system IPC improvement (percentage difference from an un-partitioned LLC), average normalized throughput of the slower application in each mix (the so-called slow-core is the workload that completes warm-up and simulation *only* once), and slow-core fairness (IPC normalized to IPC observed when the same workload is simulated alone, also referred to as weighted IPC). We also analyze best to worst case normalized throughput and fairness with percentile 1–99% of each metric. Percentiles indicate values found in a data set that exceed a designated percentage of all values in that set (i.e., P=1% yields a value that is greater than 1% of all values), and are color coded in the figure (for example, P=1% or p01 is yellow). We discuss the results in Figure 7, analyzing each column respective of each of the following sections.

5.2.1 Sweeping how we Collect Theft-based Contention. We compare the following configurations of ACE in this section: the full configuration that we detail in Section 2; allowing ACE sampling probabilistic-ally via PSA (PSA[ACE]); and a lightweight or *lite* variant of PSA[ACE] that does not store bits with each block to maintain theft tracking fidelity (PSA[ACE-lite]). Results are in the left column of Figure 7. We see that PSA[ACE-lite] has the best performance for the system, the slow-core throughput, and slow-core fairness and indicates we can use a statistics accounting framework in CASHT that has a nominal impact on added overhead. We see that raw accounting in ACE may contribute to CASHT miss-predicting partitioning solutions. Given the theft estimate sensitivity to partition configuration we discuss in Section 3, it makes sense that PSA[ACE] does slightly better for the system but worse for slow-core throughput and fairness. We attribute this to integrating the theft accounting and probabilistic sampling, which PSA[ACE-lite] does away by assuming the infrequency of sampling makes true theft accounting with a theft bit unnecessary. We assume PSA[ACE-lite] as the default statistics collection mechanism in CASHT.

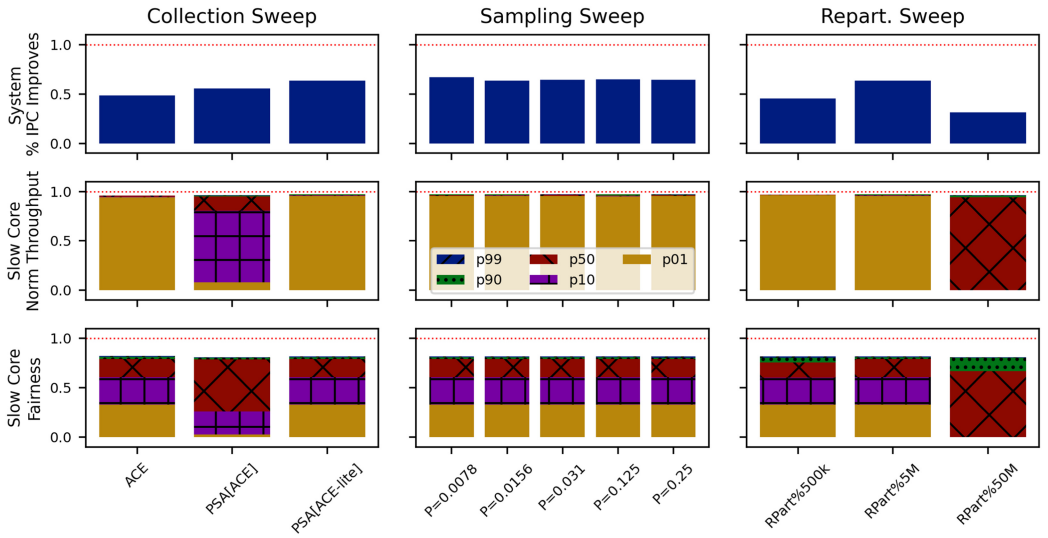


Fig. 7. Architectural Tuning: We demonstrate the results of tuning different features of CASHT, including (left) contention collection methods, (center) sampling rate, and (right) re-partitioning rate. The top row shows system performance improvement, the center row shows the percentile analysis of slow core normalized throughput where colors represent percentile value (for example, p99 normalized throughput is higher than 99% of all throughput the listed configuration achieves); (left) we see PSA[ACE-lite] is the best configuration for contention collection; (center) we see a lower sampling probability has a slight advantage; (right) a 5M cycle re-partition rate has a clear advantage.

5.2.2 Sweeping Sample Size. Sweeping Sample Probability means changing the fraction of cache we want to sample, as we discussed in Section 3. The sampling probabilities we simulate are 0.78%, 1.56%, 3.1%, 12.5%, and 25%. Results are in the center column of Figure 7. We see that each configuration is fairly similar, with 0.75% sampling having a slight advantage in the system throughput. We attribute the performance improvement at a lower sampling rate to PSA. Sampling workload accesses rather than hoping workloads access some designated allows CASHT to sample access infrequently while achieving a performance advantage. We assume 0.78% as the default sampling rate in CASHT.

5.2.3 Sweeping Re-partitioning Frequency. Sweeping the re-partitioning frequency investigates how much time passes between calls to tree-scaling and a new cache-partition allocation is determined. The set of re-partitioning time quanta we evaluate includes 500,000 (500K), 5Million (5M), and 50Million (50M) cycles between calls to tree-scaling. It is clear from the right column in Figure 7 that re-partitioning every 5M cycles has an advantage over the faster and slower re-partitioning frequencies, so we assume 5M cycles as the re-partitioning frequency for CASHT.

5.2.4 Comparing Different GBT Models at Run-time. Gradient-boosting Trees promises accurate multi-label prediction and shows high accuracy with just last-level cache features (Figure 5). Indeed, the model accuracy is similar across different features sets, and we test the following key feature sets at run-time: GBT with all features; GBT with features from LLC only; and GBT without theft-based features. We find that a GBT model trained on LLC features alone has a performance advantage (1.007 versus 1.006 when comparing normalized throughput, and 0.99 versus 0.98 when comparing fairness) and suggests core features have a normalizing impact on the partition predictions at run-time. Further, we find there is a tradeoff between the LLC-only GBT model that

includes theft-based features and a model that does not include these features. All models are within a few percentage points of performance when comparing the best performing values (percentile 90 and above), but using the GBT model trained on LLC features (including those that are theft-based) does less harm to the worst performing mixes. We refer to the configuration that employs GBT trained only on LLC features with theft-based features as CASHT for the remainder of this work given this finding.

6 EXPERIMENTAL SETUP

6.1 Simulator

We use ChampSim [22], from the second cache replacement competition [1], as our simulation environment, and we modify it to allow dynamic re-partitioning and embedded python calls for the GBT model. ChampSim is trace-based, cycle-approximate, and simulates more than one workload trace such that each workload completes a set number of instructions. We configure Last-level Cache to be 16 way set associative, with cache capacity per core set at 2 MB with 64 B blocks. As noted by FIESTA [17], trace-based simulation can take two forms: fixed work where each trace only completes the same amount of work; and variable work where the total number of instructions to simulate is set and each trace runs until this goal is reached. ChampSim follows the fixed work method by warming cache for the first N instructions and simulating for the next M instructions; however, for *cores* >1, warm-up and simulation completes when both workloads complete N and M instructions, respectively. If one workload completes before the other, then that workload restarts from the beginning of the traces. Due to simulator behavior, we focus performance analysis on the trace per each pair that completes once and identify it as *Slow-core* or *Latency Critical* workload throughout our analysis. Additionally, please note that this version of ChampSim has an eight-core upper bound on the number of cores it supports.

6.1.1 Learning Model Integration. We built the GBT model in python as we described in Section 4. We train the GBT model with data we collect through exhaustive simulation of each variation of dividing cache ways between two traces, and the two traces are selected from a list of unique pairings of the SPEC-17-based traces we list in Table 3. We embed a python interpreter into the C/C++-based simulation environment to use GBT via tree-scaling at run-time. A trained GBT model is saved offline via the pickle package, and the tree-scaling function loads/unloads the model at each re-partition call in our modified version of ChampSim.

6.1.2 Policies. We compare dynamic CASHT+GBT against UCP, a static and even partition allocation (EvenPart, or Even), and a static oracle partition that we compose from exhaustive partition simulations (Static Oracle or S.Orcle). We assume way-based partitioning similar to **Intel Cache-partitioning Technology (Intel CAT)** [15] as the partitioning scheme and full Least Recently Used as the replacement policy for all of the techniques. Physical way partitioning has some caveats like so-called block orphaning where a live block could be left out of the partition of the workload that initially requests it once a re-partitioning step occurs. We do not address this issue in either CASHT or UCP, but static solutions do not have this problem. We also recognize there exist numerous partitioning schemes in the literature, but recent works employ partition clustering, which we do not [11], or are security-minded, which we are not [23]. There are partitioning schemes that we exclude from the comparison the cache architecture (z-cache [34]) does not exist in commodity systems [3, 25, 43]. Last, results present in CASHT were generated via the Open Source Grid [28, 36] and the Tufts High Performance Cluster [44].

Table 3. SPEC 17 Trace Characteristics Bold names indicate LLC Intense workloads (L2 MPKI > 5 LLC APKI > 5)

Benchmark	Footprint (kB)	LLC APKI	LLC MPKI	WSS_{Mean}(kB)	WSS_{StdDev} (kB)	$WSS_{variance}$ (kB)
500.perlbench	1.024	2.095	0.639	1.393	2.062	4.252
502.gcc	31.488	2.214	0.785	2.891	2.203	4.853
503.bwaves	959.808	43.83	39.407	199.611	207.06	42,873.844
505.mcf	6.656	32.092	12.621	18.138	20.462	418.693
507.cactuBSSN	39.104	3.176	2.075	6.508	5.454	29.746
508.namd	182.208	2.106	0.282	3.145	20.292	411.765
510.parest	54.272	5.757	1.309	5.255	7.042	49.59
511.povray	45.888	0.006	0.003	0.046	0.582	0.339
519.lbm	456.960	49.558	27.646	90.965	47.702	2,275.481
520.omnetpp	22.592	4.589	2.852	13.406	21.221	450.331
521.wrf	1.024	17.823	0.042	0.357	1.078	1.162
523.xalancbmk	3.584	21.343	0.192	1.273	3.979	15.832
525.x264	1.536	0.274	0.047	0.272	0.333	0.111
526.blender	56.704	0.086	0.058	0.587	1.009	1.018
527.cam4	89.2672	10.176	5.442	31.589	142.594	20,333.049
531.deepsjeng	0.896	0.482	0.228	0.884	0.734	0.539
538.imagick	11.264	3.379	0.001	0.012	0.026	0.001
541.leela	0.384	0.061	0.003	0.038	0.038	0.001
544.nab	66.368	3.965	0.18	1.107	1.923	3.698
548.exchange2	0.448	0.0	0.0	0.0	0.014	0.0
549.fotonik3d	9.856	0.083	0.041	0.447	0.118	0.014
554.roms	28.160	40.21	16.559	47.572	64.976	4,221.881
557.xz	3.520	0.615	0.324	1.454	1.684	2.836
600.perlbench	1.216	2.05	0.641	1.244	1.921	3.69
602.gcc	1.216	2.199	0.789	2.689	2.026	4.105
603.bwaves	5.504	30.624	16.863	3.594	1.959	3.838
605.mcf	41.728	42.554	18.716	13.35	26.1	681.21
607.cactuBSSN	9.344	6.833	2.582	2.998	2.154	4.64
619.lbm	98.176	35.563	35.563	241.737	183.1	33,525.61
620.omnetpp	15.424	4.618	2.859	11.42	17.925	321.306
621.wrf	11.264	19.596	8.037	3.748	3.654	13.352
623.xalancbmk	4.864	21.343	0.194	1.297	3.322	11.036
625.x264	1.088	0.274	0.047	0.244	0.298	0.089
627.cam4	1647.808	19.193	9.317	81.221	269.089	72,408.89
628.pop2	36.416	109.498	23.125	158.976	423.598	179,435.266
631.deepsjeng	3312.320	92.169	46.071	412.726	725.777	526,752.254
638.imagick	78.208	5.299	3.463	1.922	0.792	0.627
641.leela	0.576	0.055	0.003	0.039	0.038	0.001
644.nab	313.856	0.184	0.092	0.828	0.147	0.022
648.exchange2	0.384	0.0	0.0	0.0	0.012	0.0
649.fotonik3d	31.104	0.101	0.047	0.457	0.101	0.01
657.xz	104.064	260.872	173.913	716.056	496.498	246,510.264

Note: footprint (kB) = (# unique 64 B block addresses)/1,000; Working set size (WSS) = mean((# unique 64 B block addresses per 250k instructions)/1,000)).

6.2 Benchmarks

Our workloads are traces we generate by skipping the first 1B instructions of each benchmark in SPEC 17 [40] and tracing the following 750M instructions. The trace characteristics are shown in Table 3 (LLC intense traces in bold). We warm caches with the first 500M instructions and simulate the remaining 250M instructions of each trace, a method similar to what is done in prior work [32, 45, 48]. Traces are often generated by choosing representative regions [38], but the reasons

for using representative regions are expedient characterizations of benchmarks and confidence that key parts of a trusted workload are being used to exercise the architecture. Our work is not a characterization of SPEC 2017, nor do we indicate our traces as being representative of SPEC 17 benchmarks. Our traces are important to exercise the caches and DRAM enough to produce diverse behavior across our mixes, and the amount of experiments we derive from all unique pairings of traces provides us with such variety. Finally, mix generation is exhaustive for the two-core simulations (totaling 860 mixes), while the four- and eight-core mixes are randomly generated with guarantee of at least 1 LLC intense workload per mix. In the end, we have 106 four-core mixes and 45 eight-core mixes.

6.3 Performance Equations and Analysis

Metrics to evaluate performance for partitioning techniques are normalized throughput and fairness. We calculate normalized throughput as $IPC_{configuration}/IPC_{LRU}$, where results greater than 1 indicates an improvement in throughput while those less than 1 indicates performance loss. Fairness can be represented as weighted IPC, or $IPC_{c,configuration}/IPC_{iso}$, where c indicates a workload in a C-workload mix ($C > 1$) and iso indicates IPC for workload c when run alone. Fairness is often referred to as weighted IPC.

7 PERFORMANCE ANALYSIS

We study CASHT in two-, four-, and eight-core configurations in this section. In the two-core analysis, we compare CASHT to UCP and two static partitioning solutions: Even or EvenPart, which is a naive, equal partitioning solution; and Static Oracle or S.Oracle, which we choose manually by inspecting all two-core partitioning configurations and choosing the configuration that maximizes system throughput. Static policies provide a known floor and ceiling to partitioning performance that we can consider the re-partitioning solutions within, and having the static oracle simultaneously allows us to understand how far the CASHT strays from those solutions. In the four- and eight-core analysis, we compare CASHT and UCP to illustrate how well the tree-scaling algorithm enables CASHT to approach UCP performance with a fraction of the overhead. For our performance analysis, we use the normalized throughput and fairness metrics described in Section 6, and we refer to these measures as such for the rest of the article.

7.1 Two-core Analysis

We compare s-curves for Static Oracle, UCP, Even Partitioning, and CASHT in Figure 8. We plot s-curves (performance metric sorted from smallest to largest normalized throughput) and the average of those results for all trace in each of the 860-trace pairs. Because it is difficult to see all 860 results in an S-curve, we have broken curves to zoom into the interesting ends: the throughput results where the static oracle loses at least 0.5% from the unpartitioned case (or normalized IPC <0.995; totals 87 data points); and the throughput results where the static oracle gains at least 0.5% over the unpartitioned case (or normalized throughput >1.005; totals 240 data points). The top row shows normalized throughput for the traces and the bottom row shows fairness for the traces. In summary, CASHT averages 0.57% improvement in throughput and does no more than 1.8% harm to average single trace performance (i.e., averages 0.982% for fairness). While CASHT does not achieve the 1% average improvement in *Latency Critical* Trace throughput that the Static Oracle achieves, CASHT is within the margin of error of UCP performance at 1/8 the overhead in the two-core configuration.

7.1.1 Two-core Throughput. CASHT improves throughput over LRU by 0.57% on average across 860 two-trace experiments, improves as much as 60% in the best case, and harms throughput well

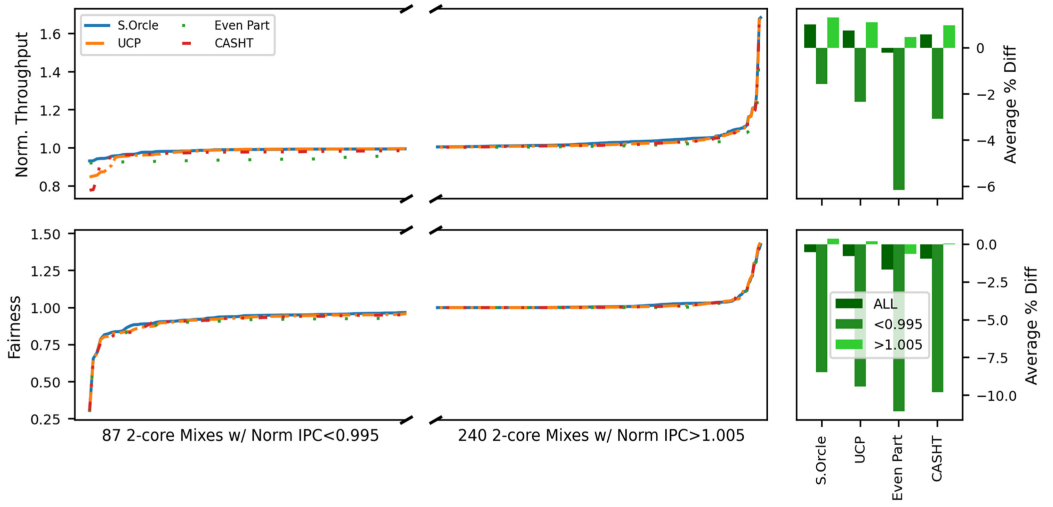


Fig. 8. CASHT Two-core Throughput and Fairness: We compare two-core CASHT against Utility-based Cache Partitioning, Even Partitioning, and the Static Oracle Partitioning solution. Given that we have 860 experimental results, we break the S-curves to make the end results easier to see, showing ranges of performance that saw >0.5% change in throughput on the left and show summaries for these ranges per configuration on the right; (top) S-curves indicate CASHT trades (huge) overhead for a marginal loss in throughput; (bottom) S-curves indicate CASHT is less fair than UCP, which is reflected in the average fairness, though the average results for CASHT and UCP are still within the margin of error.

within a noise margin of LRU in the worst case. By comparison, UCP has similar wide extremes within the data that comprise the average throughput improvement over LRU, achieving a max 75% improvement in the best case and a −20% in the worst case. It is clear that CASHT has comparable performance and behavior to UCP due to the similarity performance across the range of two-core simulations. We would like to note that CASHT-full (CASHT with GBT trained on the full cache hierarchy) can exceed the oracle in the absolute worst case range (furthest left), but requires core cache information to do so. A future version of CASHT could take advantage of core hints rather than full core cache statistics to minimize cost.

7.1.2 Two-core Fairness. Similar to the throughput analysis, we study the fairness S-curves and average percentage change in fairness of each configuration. We use weighted IPC as a proxy for fairness, or a measure of how much impact (positively or negatively) sharing cache has on the performance of a trace when run alone (or single-trace IPC). CASHT achieves a fairness of 0.982 on average, which translates to a −1.8% loss in IPC versus single trace IPC, has a worst-case fairness of 0.25, which translates to a 75% loss, and best case weighted IPC of 1.48, which translates to 48% gain in IPC. UCP has similar high marks in fairness but does better in the worst case, which translates to an average weighted IPC of 0.996 or −0.6% loss in IPC versus single trace IPC. The Even and Static Oracle partitioning solutions frame the dynamic policies at the bottom and top of average performance, respectively.

7.2 Percentile Analysis

We analyze the performance impact each technique has on individual traces by analyzing the percentiles for individual workload performance while sharing LLC. Percentiles (P) indicate a value

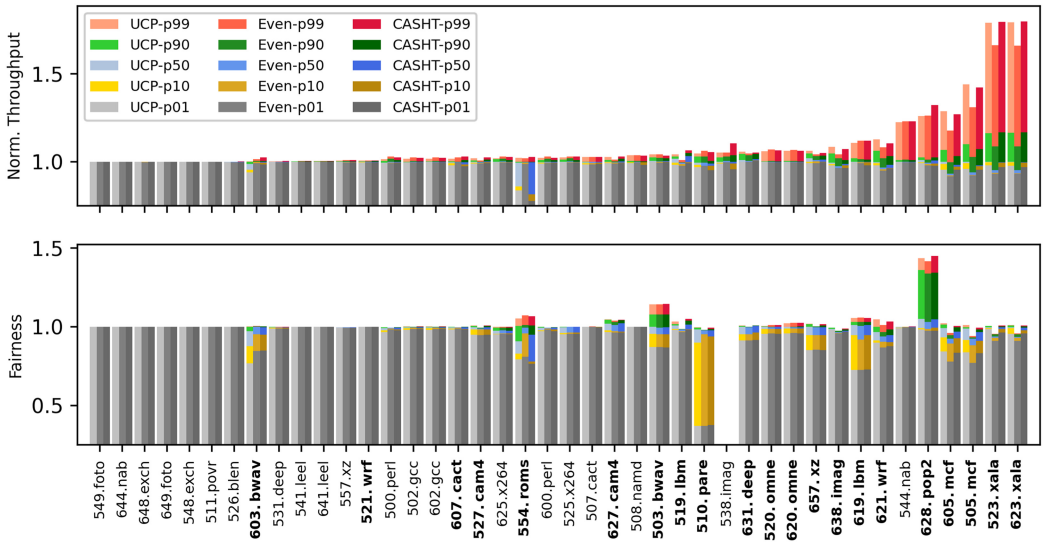


Fig. 9. Layered Throughput and Fairness Two-core Percentiles: We study the percentiles of normalized throughput (top) and fairness for workloads when they are *Latency Critical* (or slowest to complete) (bottom) to understand the impact of UCP, Even Partitioning, and CASHT in both contexts; x-axis shows benchmarks sorted by UCP p99 throughput values, y-axis for the top is throughput (IPC_{cfg}/IPC_{LRU}), and y-axis for the bottom is fairness (IPC_{cfg}/IPC_{Iso}); techniques are distinguished by color density (light is UCP, medium is Even, dark is CASHT), and percentiles are distinguished by color families (p99=red, p90=green, p50=blue, p10=yellow, p01=gray) and are layered from P99 to p01 for a compact representation of all percentiles; Bold names indicate LLC intense workloads. **NOTE: 538.imagick has no fairness data, because it never runs only once for proper normalization to the Iso case**; (top) we see UCP has clear advantages for mcf- and xalancbmk-based traces (among others); CASHT does not reach the heights of UCP, but is comparable at 1/8th the size; (bottom) UCP; (bottom) shows a CASHT and UCP are largely similar while outperforming Even partitioning, with a few benchmarks with lower p50 for CASHT vs. UCP, and one instance (603.bwaves & 554.roms) where CASHT has higher percentile values than UCP.

(V) in a data set for which N% of all values are less than V. Figure 9 shows P=1%, P=10%, P=50%, P=90%, and P=99% of normalized throughput and fairness for each trace when in a shared cache. The x-axis shows benchmarks sorted by UCP p99 throughput values, the y-axis for the top reflects throughput (IPC_{cfg}/IPC_{LRU}), and y-axis for the bottom reflects fairness (IPC_{cfg}/IPC_{Iso}). Each technique is distinguished by color density (light is UCP, medium is Even, dark is CASHT), and percentiles are distinguished by color families (p99=red, p90=green, p50=blue, p10=yellow, p01=gray), which are layered for compact representation of all percentiles per configuration. In summary, CASHT does not reach the peak performance of UCP but has a higher lower percentile indicating less harm to the worst 1% of normalized throughput. UCP has a clear advantage for mcf- and xalancbmk-based traces (20–79% gains in normalized throughput over LRU). Additionally, UCP also has performance advantages for 621.wrf, 638.imagick, and 657.xz. CASHT has some advantage in 500.perlbench, 510.parest, 603.bwaves, 628.pop2, and 619.lbm, though we can attribute some of the advantages to evenly splitting the cache between traces given that Even Partitioning has similar or better solutions in most of these cases. On close inspection of the p10 and p01 values, we observe CASHT has the advantage in minimizing harm for many LLC intense workloads (in bold in Table 3), indicating CASHT does less harm when LLC is more intensely in use. Add to this

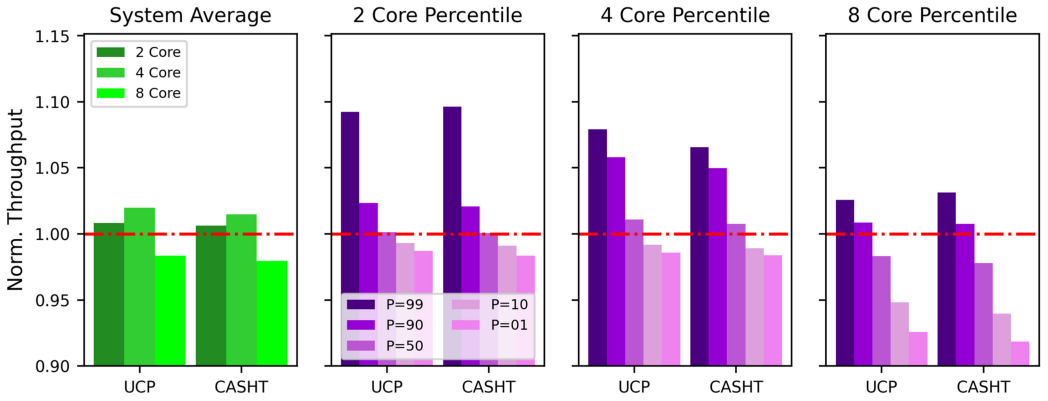


Fig. 10. Two-, Four-, and Eight-core Performance Summary: We show summary normalized throughput data for the two-core (860 simulations), four-core (106 simulations), and eight-core (45 simulations) results for UCP and CASHT. The y-axes represent normalized throughput (IPC_{cfg}/IPC_{LRU}); system average represents the average across all experiments per configuration; Two-core Percentile represents the performance metric thresholds where N% of all performance results are less than the shown value (Four- and Eight-core Percentile is similar, respective of the core configurations). The left plot shows CASHT approaches UCP on average as we scale cores from two to eight cores. The percentile plots simultaneously show the range and composition of impact that UCP and CASHT have on performance is similar.

the fact that a >50% of traces have fairly similar results, and the attraction of a lighter technique is evident in those cases. Indeed, CASHT approaches UCP peak performance and minimizes harm to worst-case throughput at 1/8 the cost.

7.3 Core Scaling

We analyze average normalized throughput and fairness metrics for UCP and CASHT when scaled from two to eight cores in Figure 10, and it is clear that CASHT approaches UCP performance in all metrics at each core configuration. The y-axis represents normalized throughput while the x-axes show each partitioning configuration. Each plot reflects different presentations of average performance data from left to right: the first plot shows the average performance of the system; the second shows the performance metric percentile values from 99 to 1 at a two-core configuration; the third shows the performance metric percentile values at a four-core configuration; and the fourth shows similar percentile results but for the eight-core configuration. Percentiles indicate a value in a data set that exceeds the ascribed percentage of that data set (for example, p=10 throughput is greater than 10% of all throughput values in a common set of data). We see that UCP and CASHT have comparable average throughput from the first plot, with CASHT having the advantage due to being a fraction of the overhead of UCP. Further, the percentile plots (plots 2 through 4) support that the lightweight CASHT framework not only approaches the heavyweight UCP implementation in the performance yielded per percentile but also approaches similar performance at a larger core count without the steep cost of scaling (aside from the additional hardware counters for thefts and interference that each core requires).

8 RELATED WORK

8.1 Cache Contention Measurement

Modeling cache events and cache behavior are the foundation for performance analysis tools and efficient architectures. We compare related events and models to our theft-based metrics in this

section. The **Higher Order Theory of Locality (HOTL)** describes **footprint (fp(x))** as the number of unique accesses in a window of accesses across a trace, and this footprint can be used to derive miss rate and reuse distance [47]. The fp(x) metric necessitates time-based sampling to capture correctly, which likely results in occupying cache with this task and harming performance. Further, there is abstract knowledge of inter-core impact on footprint and derived metrics rather than direct knowledge (like with thefts). **Average Eviction Time (AET)** models least recently used stacks on a byte granularity to approximate the miss rate curve [27]. AET suggests periodic computation similar to what is done in many partitioning algorithms (including CASHT), but approximates miss rate and not cache contention behavior. Flow, or the rate that cache blocks are moved toward eviction, acts as a proxy for miss rate in Whirlpool [25] and is leveraged to approximate combined miss curves, which enables clustering in Kpart [11]. By comparison, our theft-based metrics are not models but actual events that are consequences of workload interaction and require we identify cause and effect directly. Flow is often taken in isolation and is also used as a proxy for misses, but it is well known that misses are not representative of performance in a deeper cache hierarchy (except in the extreme cases). Thefts show significant correlation to performance, are shown to complement miss-based statistics in Section 2, and can be used in concert with the above models.

8.2 Partitioning

The techniques in Table 4 show a range of applications and implementations of cache repartitioning in recent literature. We compare each with each other and our technique, CASHT across the following technique features: partition allocation **algorithm**; whether partitions are split between cores (**C**) or threads (**T**); what cache dimension (set or way) are caches **partitioned**; how partitions are **enforced**; if they are hardware (**hw**) or software (**sw**)-based; how cache behavior is **profiled**; when **re-partitioning** occurs; and the **overhead**. UCP tracks per workload cache hit curves to compute and balance the cache needs every 5M cycles [32]. UCP introduced the lookahead algorithm to cache-partitioning problem, and many works can and do adopt the algorithm as a foundation in their work [35, 45, 48], but UCP has large overhead and does not scale well as core counts scale up. COLORIS leverages page coloring via custom page allocator to partition cache along sets [49], but requires modifications to the operating system. Kpart exploits way partitioning adoption in silicon to partition last-level cache into cluster groups, and uses IPC (plus miss rates and bandwidth) to cluster workloads before passing input to the lookahead algorithm [11]. Kpart clusters applications to avoid the diminished returns of partitioning a few ways between many cores, which is not the goal of CASHT. Further, Kpart without clustering is similar to UCP adapted in software given that the lookahead algorithm is in use to determine partition sizes at each repartition step, so we believe comparing against UCP is sufficient. Cooperative partitioning addresses orphaned lines and smooth the transitions after a re-partition step occurs, and modifies lookahead to allow for blocks to be disabled altogether [42]. **Reuse locality aware cache algorithm (ROCA)** partitions between reused and not-reused cache blocks, leveraging a reuse predictor to determine partition placement at insertion [37]. This differs from the approach taken by partitioning algorithms generally, but can be reduced to identifying blocks by prediction rather than CPU ID so most way-based can adapt to this problem. **Gradient-based Cache-partitioning Algorithm (GPA)** enables continuous partition adjustment at a cache block granularity by determining how the rate at which blocks are placed in cache in a protected (or vulnerable) state [13]. Consequently, the usage of gradient sets can cause harm to a portion of cache due to purposeful beneficial and detrimental behavior across gradient sets, which CASHT avoids with PSA (Section 3). **Machine learned cache (MLC)** partitions L2 AND L3 cache via a trained reinforcement learning model called Q-learning, enables smart and collaborative partitioning decisions between per thread Markov agents [18]. Though MLC and CASHT both

Table 4. Last-level Cache-partitioning Framework Comparison

Framework	Algorithm	C or T	Partition	Enforce	hw or sw	Profile	Repart.	Overhead
UCP	LA	C	W	W-based	hw	samp	cyc	3.7 kB/C
COLORIS	recolor engine	T	S	pg color	sw	$umon	thrsh	mod pg-allc
KPART	cluster+LA	T	W	W mask	sw	dynaway	cyc	$O(A^2W^2)$ lat
Cooperative	mod LA	C	W	W mask	hw	samp	cyc	4.1 kB/C
ROCA	blk-migr.+LA	—	W	W mask	hw	samp	cyc	82 kB
Gradient	hill climb	T	blk	statistical	hw	gradient set	always	5 B/T
MLC	Q-learn	T	W	W mask	hw	agent/T	cyc	875 B
CASHT	GBT+tree scale	C	W	W-based	hw	PSA	cyc	16 B/C+1 kB

Assumes 16-way, 4 MB LLC *Key*: LA=lookahead; $=cache; hw=hardware; sw=software; cyc=cycles; A=applications; C=core; T=thread; W=way; S=set; blk=block; umon=utility monitor; samp=sampler; mod=modify; pg=page; thrsh=threshold; allc=allocator; lat=latency; GBT=gradient-boosting trees (Section 4).

take advantage of learning algorithms, MLC partitions both L2 and L3 to achieve performance gain on a system that assumes Simultaneous Multi-Threading, which CASHT does not.

8.3 Summary

Theft-based metrics offer significant and complementary performance correlation, enable run-time contention analysis with the addition of two hardware counters per core or thread, and the theft mechanism allows estimation in the face of partitioning. Miss-based metrics, which are often collected in isolation, do require added overheads like a set sampler or run-time phases where application performance is harmed to collect them. Further, given that LLC misses (especially taken in isolation) are frequently reported as misleading, models based on such behavior render partial information and theft-based metrics can fill those gaps.

CASHT leverages theft-based metrics toward to address the cache-partitioning problem by enabling run-time contention analysis and coupling the results with a supervised learning model to make partitioning predictions. Prior art partitions along different cache dimensions (set or way) or employ different algorithms, but none consider cache contention directly. Additionally, the CASHT framework does not require the cache to operate in any harmful state for the sake of statistical analysis. Last, the CASHT framework approaches comparable performance to a technique with 8X the overhead for a 4 MB, 16-way LLC.

9 CONCLUSION AND FUTURE WORK

We present CASHT, a contention analysis and re-partitioning framework that enables lightweight cache partitioning within performance noise margins of the well-regarded Utility-based Cache Partitioning at a fraction of the overhead. The GBT model we train achieves 90% pseudo oracle prediction accuracy at 100 B and 95+% accuracy at 1 k+B, and the Tree-scaling algorithm allows us to scale our solution above two-core architectures. Contention estimation and lightweight sampling enabled by our ACE and PSA techniques allow us to keep overhead small enough to be nominal in comparison to UCP. Our two-core results show we are within the margin of noise (<0.5%) of UCP in both Throughput and Fairness metrics, and have room to grow in comparison to the static oracle performance we train our GBT model on. Similarly, the four-core results we are also within the margin of noise of UCP performance, affirming that the Tree-scaling algorithm is effective at scaling our two-core solution up to four cores. For future work, we will re-train GBT on run-time oracle solutions rather than static solutions. We know a prior work clusters workloads to reduce the number of partitions at core counts greater than two, so we will apply the CASHT framework toward partition clustering and compare directly to KPart. We also wish to apply novel Tree-scaling optimizations that leverage the pseudo-oracle prediction for other cache management decisions,

like changing the inclusion property for allocation predictions that indicate a workload operates best with the smallest allocation. Last, we will conduct a hyper-parameter space exploration for Tree-scaling to study the limits of our algorithm.

REFERENCES

[1] Texas A&M. 2017. Cache Replacement Championship 2. Retrieved from http://crc2.ece.tamu.edu/.

[2] Helen Akpan, Akpan, B. Rebeccajeya, and Vadhanam. 2015. A survey on quality of service in cloud computing. *Int. J. Comput. Technol. Trends* 27 (Oct. 2015), 58–63.

[3] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE, 213–224.

[4] Rebecca Belshe, Peter Schlichting, and Gil Speyer. 2020. Refactoring a statistical package for demanding memory loads: Adapting R for high performance telemetry data analytics. In *Proceedings of the Conference on Practice and Experience in Advanced Research Computing (PEARC'20)*. ACM, 444–447. https://doi.org/10.1145/3311790.3399624

[5] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*. IEEE, 318–329. https://doi.org/10.1109/MICRO.2008.4771801

[6] R. Blahut. 1974. Hypothesis testing and information theory. *IEEE Trans. Info. Theory* 20, 4 (July 1974), 405–417. https://doi.org/10.1109/TIT.1974.1055254

[7] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM, Article 52, 12 pages. https://doi.org/10.1145/2063384.2063454

[8] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 785–794.

[9] Michael Cusumano. 2010. Cloud computing and SaaS as new computing platforms. *Commun. ACM* 53, 4 (2010), 27–29.

[10] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T. Chong. 2017. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in NVMs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, 232–244. https://doi.org/10.1145/3123939.3124548

[11] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez. 2018. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 104–117.

[12] Jerome H. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *Ann. Statistics* (2001), 1189–1232. https://doi.org/10.1214/aos/1013203451

[13] William Hasenplaugh, Pritpal S. Ahuja, Aamer Jaleel, Simon Steely Jr., and Joel Emer. 2012. The gradient-based cache partitioning algorithm. *ACM Trans. Archit. Code Optim.* 8, 4, Article 44 (Jan. 2012), 21 pages. https://doi.org/10.1145/2086696.2086723

[14] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. Retrieved from http://arxiv.org/abs/1803.02329.

[15] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, 657–668.

[16] Leticia Hernando, Alexander Mendiburu, and Jose A. Lozano. 2018. Hill-climbing algorithm: Let's go for a walk before finding the optimum. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'18)*. 1–7. https://doi.org/10.1109/CEC.2018.8477836

[17] Andrew Hilton, Neeraj Eswaran, and Amir Roth. [n.d.]. FIESTA: A Sample-Balanced Multi-Program Workload Methodology. https://scholarworks.umass.edu/dissertations/AAI8509594/.

[18] R. Jain, P. R. Panda, and S. Subramoney. 2017. A coordinated multi-agent reinforcement learning approach to multi-level cache co-partitioning. In *Proceedings of the Design, Automation, and Test in Europe Conference Exhibition (DATE'17)*. 800–805.

[19] Aamer Jaleel. 2007. Memory characterization of workloads using instrumentation-driven simulation a pin-based memory characterization of the SPEC CPU 2000 and SPEC CPU 2006 benchmark suites. VSSAD Technical Report.

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

[21] Samira Manabi Khan, Yingying Tian, and Daniel A. Jiménez. 2010. Sampling dead block prediction for last-level caches. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE, 175–186. https://doi.org/10.1109/MICRO.2010.24

[22] Jinchun Kim. 2017. ChampSim. Retrieved from https://github.com/ChampSim/ChampSim.
[23] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 974–987. https://doi.org/10.1109/MICRO.2018.00083
[24] Dongsheng Liu, Zilong Liu, Lun Li, and Xuecheng Zou. 2016. A low-cost low-power ring oscillator-based truly random number generator for encryption on smart cards. *IEEE Trans. Circ. Syst. II: Express Briefs* 63, 6 (2016), 608–612. https://doi.org/10.1109/TCSII.2016.2530800
[25] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2016. Whirlpool: Improving dynamic cache management with static data classification. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, 113–127. https://doi.org/10.1145/2872362.2872363
[26] Kunle Olukotun, Lance Hammond, and James Laudon. 2007. Chip multiprocessor architecture: Techniques to improve throughput and latency. *Synth. Lect. Comput. Architect.* 2, 1 (2007), 1–145.
[27] Cheng Pan, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. 2021. Penalty- and locality-aware memory allocation in redis using enhanced AET. *ACM Trans. Storage* 17, 2, Article 15 (May 2021), 45 pages. https://doi.org/10.1145/3447573
[28] Ruth Pordes, Don Petravick, Bill Kramer, Doug Olson, Miron Livny, Alain Roy, Paul Avery, Kent Blackburn, Torre Wenaus, Frank Würthwein, Ian Foster, Rob Gardner, Mike Wilde, Alan Blatecky, John McGee, and Rob Quick. 2007. The open science grid. *J. Phys. Conf. Ser.* 78 (2007). 012057. https://doi.org/10.1088/1742-6596/78/1/012057
[29] Thomas Roberts Puzak. 1985. Analysis of cache replacement-algorithms.
[30] Yuanzhuo Qu, Bruce F. Cockburn, Zhe Huang, Hao Cai, Yue Zhang, Weisheng Zhao, and Jie Han. 2018. Variation-resilient true random number generators based on multiple STT-MTJs. *IEEE Trans. Nanotechnol.* 17, 6 (2018), 1270–1281. https://doi.org/10.1109/TNANO.2018.2873970
[31] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, 381–391. https://doi.org/10.1145/1250662.1250709
[32] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 423–432.
[33] Shenyuan Ren, Ligang He, Junyu Li, Zhiyan Chen, Peng Jiang, and Li Chang-Tsun. 2019. Contention-aware prediction for performance impact of task co-running in multicore computers. *Wireless Networks* (Feb. 2019). Springer, 1–8. https://doi.org/10.1007/s11276-018-01902-7
[34] D. Sanchez and C. Kozyrakis. 2010. The ZCache: Decoupling ways and associativity. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 187–198. https://doi.org/10.1109/MICRO.2010.20
[35] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. *SIGARCH Comput. Archit. News* 39, 3 (June 2011), 57–68. https://doi.org/10.1145/2024723.2000073
[36] Igor Sfiligoi, Daniel C. Bradley, Burt Holzman, Parag Mhashilkar, Sanjay Padhi, and Frank Wurthwein. 2009. The pilot way to grid resources using glideinWMS. In *Proceedings of the WRI World Congress on Computer Science and Information Engineering*. 428–432. https://doi.org/10.1109/CSIE.2009.950
[37] Fanfan Shen, Yanxiang He, Jun Zhang, Qingan Li, Jianhua Li, and Chao Xu. 2019. Reuse locality aware cache partitioning for last-level cache. *Comput. Electr. Eng.* 74 (2019), 319–330. https://doi.org/10.1016/j.compeleceng.2019.01.020
[38] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *ACM SIGARCH Computer Architecture News*, Vol. 30. ACM, 45–57.
[39] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. ACM, 413–425. https://doi.org/10.1145/3352460.3358319
[40] Standard Performance Evaluation Corporation. [n.d.]. SPEC Benchmark Suite. Retrieved from http://www.spec.org.
[41] Le Sun, Jaipal Singh, and Omar Khadeer Hussain. 2012. Service level agreement (SLA) assurance for cloud services: A survey from a transactional risk perspective. In *Proceedings of the 10th International Conference on Advances in Mobile Computing and Multimedia (MoMM'12)*. ACM, 263–266. https://doi.org/10.1145/2428955.2429005
[42] Karthik T. Sundararajan, Vasileios Porpodas, Timothy M. Jones, Nigel P. Topham, and Björn Franke. 2012. Cooperative partitioning: Energy-efficient cache partitioning for high-performance CMPs. In *Proceedings of the IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–12.
[43] P. Tsai, N. Beckmann, and D. Sanchez. 2017. Jenga: Software-defined cache hierarchies. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 652–665.
[44] Tufts University. [n.d.]. Tufts High-performance Computing Research Cluster. Retrieved from https://it.tufts.edu/hpc.
[45] Ruisheng Wang and Lizhong Chen. 2014. Futility scaling: High-associativity cache partitioning. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 356–367.

[46] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, 430–441. https://doi.org/10.1145/2155620.2155671

[47] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: A higher order theory of locality. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. 343–356.

[48] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, 174–183. https://doi.org/10.1145/1555754.1555778

[49] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, 381–392. https://doi.org/10.1145/2628071.2628104

[50] Po-Shao Yeh, Chih-An Yang, Yi-Hong Chang, Yue-Der Chih, Chrong-Jung Lin, and Ya-Chin King. 2019. Self-convergent trimming SRAM true random number generation with in-cell storage. *IEEE J. Solid-State Circ.* 54, 9 (2019), 2614–2621. https://doi.org/10.1109/JSSC.2019.2924094

Received June 2021; revised September 2021; accepted October 2021