Effective Cache Apportioning for Performance Isolation Under Compiler Guidance

Bodhisatwa Chatterjee  
Georgia Institute of Technology  
Atlanta, USA  
bodhi@gatech.edu

Sharjeel Khan  
Georgia Institute of Technology  
Atlanta, USA  
smkhan@gatech.edu

Santosh Pande  
Georgia Institute of Technology  
Atlanta, USA  
santosh.pande@gatech.edu

Abstract
With a growing number of cores per socket in modern datacenters where multi-tenancy of a diverse set of applications must be efficiently supported, effective sharing of the last level cache is a very important problem. This is challenging because modern workloads exhibit dynamic phase behaviour - their cache requirements & sensitivity vary across different execution points. To tackle this problem, we propose Com-CAS, a compiler guided cache apportioning system that provides smart cache allocation to co-executing applications in a system. The front-end of Com-CAS is primarily a compiler-framework equipped with learning mechanisms to predict cache requirements, while the backend consists of allocation framework with pro-active scheduler that apportions cache dynamically to co-executing applications. Our system improved average throughput by 21%, with a maximum of 54% while maintaining the worst individual application execution time degradation within 15% to meet SLA requirements.

1 Introduction
High-performance computing systems facilitate concurrent execution of multiple applications by sharing resources among them. The Last-Level Cache (LLC) is one such resource, which is usually shared by all running applications in the system. However, sharing cache often results in inter-application interference, where multiple applications can map to the same cache line and incur conflict misses, resulting in potential performance degradation. Another aspect of this application co-existence in LLC is the increased vulnerabilities to shared-cache attacks such as the side-channel attacks [9, 17, 33], timing-attacks [3, 13], and cache-based denial-of-service (DOS) attacks [2, 11, 19, 35]. Furthermore, these problems are exacerbated by the fact that the number of cores (and thus the processes) that share the LLC are rapidly increasing in the recent architectural designs. Therefore, both from a performance and security point of view, it is imperative that LLCs are carefully managed.

To address these issues, modern computing systems use cache partitioning to divide the LLC among the co-executing applications in the system. Ideally, a cache partitioning scheme obtains overall gains in system performance by providing a dedicated region of cache memory to high-priority cache-intensive applications and ensures security against cache-sharing attacks by the notion of isolated execution in an otherwise shared LLC. Apart from achieving superior application performance and improving system throughput [7, 20, 31], cache partitioning can also serve a variety of purposes - improving system power and energy consumption [6, 23], ensuring fairness in resource allocation [26, 36] and even enabling worst case execution-time analysis of real-time systems [18]. Owing to these overwhelming benefits of cache partitioning, modern processor families (Intel® Xeon Series) implement hardware way-partitioning through Cache Allocation Technology (CAT) [10]. Such systems aim to provide extended control and flexibility to the user by allowing them to customize cache partitions according to each application's requirement.

On the other hand, modern workloads exhibit dynamic phase behaviour throughout their execution resulting in rapidly changing cache demands. These behaviours arise due to complex control flows and input data driven behaviors in applications and its co-execution environments. It is often the case that even across the same program artifact (such as a loop), different cache requirements are exhibited during different invocations of the same. Majority of the prior works [6, 20, 26, 31] on cache partitioning often tends to classify applications into several categories on the lines of whether they are cache-sensitive or not. Based on this offline characterization, cache allocations for various application are decided though a mix of static and runtime policies. The fact that a single application (or even a given loop) can exhibit dual behaviour, i.e both cache-sensitive & cache-insensitive behaviour during its execution, is not taken into account. Some approaches employ a ‘damage-control’ methodology, where attempts are made to adjust cache partitions dynamically after detecting changes in application behaviour by runtime monitoring using performance counters and hardware moni-
 tors [6, 22, 28, 32]. However, these approaches are reactive and suffer from the problem of detection and reaction lag., ie. the application behaviour of modern workloads are likely to change before the adjustments are made leading to lost performance.

In order to account for applications’ dynamic phase behaviors and provide smart & proactive cache partitioning, we propose Compiler-Guided Cache Apportioning System (Com-CAS) with the goal of providing superior performance & maximal isolation to reduce the threat of shared cache based attacks. The front-end of Com-CAS consists of Probes Compiler Framework, which uses traditional compiler analysis & learning mechanisms to predict dynamic cache requirements of co-executing applications. The attributes that determine cache requirements (memory footprint, and data-reuse) are encapsulated in specialized library markers called ‘probes’, which are statically inserted outside each an loop-nest in an application. These probes communicate the dynamic resource information to BCache Allocation Framework, which makes up the back-end of Com-CAS. It consists of allocation algorithms that dynamically apportions the LLC for co-executing applications based on aggregating probes-generated information, and a proactive scheduler that invokes the Intel CAT to perform the actual cache partitioning according to application phase changes.

We evaluated Com-CAS on an Intel Xeon Gold system with 35 application mixes from GAP [1], a graph benchmark suite, Polybench [21], a numerical computational suite, and Rodinia [5], a heterogeneous compute-intensive benchmark suite that represent a variety of workloads typically encountered in a cloud environment. The average throughput gain obtained over all the mixes is 21%, with the maximum one being upto 54% with no degradation over a vanilla co-execution environment with CFS with no cache apportioning scheme. We also show the Com-CAS maximises the isolation of reuse heavy loops providing protection against the DOS attacks simultaneously while maintaining the SLA agreement of degradation < 15%. Overall, the main contributions of the paper can be summarized as follows:

- We present a compiler-guided Cache Apportioning System that dynamically adjusts the LLC partitioning to account for changing phase behaviours of modern workloads.

- We show that cache requirements of applications can be estimated through a combination of memory footprints, data-reuse behaviour, phase-timing and cache sensitivity and we implement Probes Compiler framework that extends the traditional compiler analysis to calculate or estimate these parameters and communicate them during program execution.

- We demonstrate that smart cache apportioning can be achieved by cache allocation algorithms & proactive scheduling mechanism, implemented as BCache Allocation Framework. The framework maintains locality of partitions allocated to a given application. A net result of such apportioning decision making is that LLC misses are shifted from applications with higher cache sensitivity (high sensitivity reuse) needs to the lower ones (lower sensitivity reuse or streaming ones).

- Our results on all the benchmark suites show that superior performance of 21% average throughput increase with a maximum of 54% including enhanced security against shared-cache attacks and SLA agreement for all applications can be achieved by our system.

The remainder of this paper is structured as follows: We begin by providing a high-level overview of Com-CAS in Section 3. The prior works and their shortcomings are discussed in 4. The Probes Compiler Framework and techniques to estimate the cache requirements is presented in Section 5 and the apportioning algorithms and cache-partitioning scheme for BCache Allocation Framework are explained in 6. The experimental evaluation of our system along with detailed case studies are discussed in 7. The final remarks and conclusions are presented in Section 8.

## 2 Motivation: Dynamic Cache Apportioning

Determining memory requirements for a mix of running applications in a system is quite a complex task. The cache demand for each process is dependent on specific program points and can change drastically throughout its execution. It is possible for a same program artifact (loop) to exhibit different behaviour across multiple invocations during single execution. Identifying these dynamic variations in cache requirements of an application is paramount for apportioning cache and obtaining superior performance. For example, let us consider an application mix, consisting of 5 instances of BC benchmark from the GAP Suite [1]. These applications are provided with the same input - Uniform Random Graph with $2^{37}$ nodes. Our goal is to leverage cache partitioning through Intel® CAT to obtain superior performance. We now consider different possibilities to show that obvious solutions do not work: Can we partition the cache equally among identical processes? Since the distribution of processes in the mix are identical, it is a natural inclination to provide each process an equal amount of isolated cache ways. In a 11-way set associative cache system, one possible partition is giving any four processes 2-ways each and the fifth process 3-ways. However, as we can see in Fig. 1, following this approach results in a performance degradation by over 16%, compared to the processes running without any cache apportioning. Clearly, giving each processes 2-ways is not enough in terms of capacity. How about increasing the cache allocation to optimal ways? By performing offline profiling, the optimal number of cache-ways that minimizes execution time of an application can be figured out statically. For a single BC
application, allocating 5 cache-ways minimizes its execution time. By this logic, we can choose to give all process 5-ways each, which will be shared to some extent, owing to the 11-way limit in the current system. From Fig. 1, it is evident that this also does not work well. The reason for this is that all the 5 cache-ways are not needed by each processes throughout the entire execution period and the allocation needs to be managed accordingly. Sharing of all the ways does not provide any performance isolation since a process can potentially evict other processes data leading to performance degradation. This approach also potentially impacts security since all processes now are sharing their ways throughout the execution time. While full isolation in terms of resources might be infeasible due to resource crunch it entails, isolation can be implemented between loops which are cache sensitive giving them separate ways which is the theme of the paper. Moreover, in actual workloads, we cannot expect an identical or similar process distribution such as this example. Thus, it is evident from these attempts that statically invoking a programmable cache partitioning interface like Intel® CAT is not enough; we need to account for the changing cache requirements based on application’s execution phase for obtaining performance improvements by providing as much isolation as possible.

![Figure 1: Performance degradation by injudicious cache partitioning](image)

**3 Our Approach: Com-CAS Overview**

In order to satisfy the need for dynamic cache apportioning, we propose a compiler guided approach for solving the problem of apportioning LLC across co-executing applications on a socket. Fig. 2 depicts a high-level representation of our dynamic cache apportioning system Com-CAS. The system can be broadly divided into two major components: Probes Compiler framework, which profiles applications to determine cache requirements at loop-nest granularity & statically inserts ‘probes’ to encapsulate & broadcast the necessary information during runtime and BCAnother Cache Allocation Framework, that makes use of the probe-generated information to dynamically make apportioning & co-location decisions on a socket.

To estimate precise cache requirements, the probes generates cache memory footprints & sensitivity, data-reuse behaviour and anticipated phase duration for each loop-nest present in an application (Fig. 3). The cache footprints of a loop-nest are estimated by the polyhedral compiler analysis [8] and the expected duration is predicted by training a regression model on the loop bounds. Static compiler loop analysis based on data-reuse patterns, is performed to classify loops nests into streaming & reuse, while cache sensitivity is determined by offline profiling. Each probe encapsulates all these information and are inserted at the entrances of their respective loop nests. During runtime, the scheduler aggregates dynamic information coming from all co-executing probe-instrumented applications and invokes BCAnother Cache Allocation Framework, whose goal is to maximize isolation & enhance performance. The apportioning algorithms attempts as much isolation as possible; when sharing cache is inevitable, it aggressively performs sharing of certain kinds of loops (the ones exhibiting low data reuse) whereas minimizes sharing loops exhibiting heavy data reuse. We assume that all processes must go through Probe Compiler Framework, so the BCAnother Cache Allocation Framework can utilize its allocate ways to these processes. In addition, the mixes do not have any processes that have malicious loops or a high security application with the need of full isolation.

**4 Background & Related Work**

Cache partitioning is commonly used to divide the last-level cache (LLC) capacity among the multiple co-running applications in a system. In this section, we will present prior works on cache partitioning, focusing on both hardware-driven and software/compiler-driven approaches. We will conclude this section by giving an overview of Intel CAT, which is a re-configurable implementation of way-partitioning and is used by our system to partition the LLC.

**4.1 Cache partitioning approaches**

**Hardware-driven cache partitioning:** Cache partitioning is achieved by using specialized hardware support to physically enforce partitions in the cache. The enforced partitions are usually allocated on a granularity of cache-ways and this technique is commonly known as way-partitioning. Since sophisticated partitioning at fine granularity levels often requires extensive hardware support and appropriate software modifications to interact with the novel hardware, most research related to sophisticated way-partitioning is validated only with architectural simulators [4, 12, 25, 28]. Modern processors typically implement only basic way-partitioning due to the feasibility of such an implementation in a real hardware. The main drawback of basic way-partitioning is that one ends
Figure 2: Overview of Com-CAS consisting of Probes Compiler Framework & BCache Allocation Framework

Figure 3: Cache profile for each loop nests are generated & encapsulated in Probes

up with coarse-grain cache partitions, that is limited by the number of cache-ways in the system. Several works have been proposed to counter these limitations and obtain fine-grain partitions, such as using hash functions to access ways [24], ranking the cache line utility to decide replacements [30] or extending hardware support. Since current commodity hardware supports the basic way level partitioning, this work is based on the same.

Software-driven cache partitioning: Such techniques involves using either the compiler or the OS to restrict the memory pages into confined cache sets. The OS utilizes page-coloring [16, 27, 29, 34, 37, 38] to partition the cache, where the idea is that the physical pages are divided into distinct colors and each color can be mapped to a specific cache set. While this approach does not need additional hardware support, the main drawbacks here are that dynamically changing the partitions incurs the overhead of re-coloring the physical pages and often additional software modifications like partitioning the entire main-memory are inevitable. On the other hand, the compiler approaches uses special instructions to place different applications into different cache partitions [18, 23]. However, these techniques mostly approach cache-partitioning as a static problem and do not take into consideration dynamic loop information (such as memory footprints, cache behaviour, etc); nor they account for dynamic concurrency of different application phases to determine the dynamic cache demands. Modern applications exhibit complex dynamic phase behaviors in terms of their cache requirements as well as cache sensitivity both being different at different program points. In addition, these behaviors even change across multiple invocations of the same loop nest. The solution would involve deciding both co-location of applications on a socket as well as effective apportioning of the LLC, which is a distinct characteristic of our proposed approach.

Recent works have focused on new approaches to overcome limitations of way-partitioning and page coloring [7, 20, 31]. El-Sayed et al. [7] takes an approach of clustering a group of applications based on their isolated and combined miss-curves and partitioning the cache among them. This approach requires profiling every pair of applications in the mix in addition to profiling them separately, i.e performing $O(n^2)$ profiling operations on top of $O(n)$ individual operations. Apart from being non-scalable for mixes containing large number of applications, this approach is not practical in real-world scenarios like data centers, where mix composition or even participating processes therein are not even known apriori. Moreover, in the above approaches, the proposed cache partitions “plans” are adjusted over fixed intervals - the approaches do not account for the dynamic phase changes in each application. An alternative approach that accounts for application phase changes is proposed by Pons et al. [20], wherein based on the execution behavior, applications are categorized into multiple categories by extensive profiling. However, such static categorization misses an important aspect - application phases are input-dependent and are dynamic as well as contingent to entire mix composition. Thus, an application behaviour might change based on an different execution scenario and a new input data set. Moreover, the approach misses accounting for dynamic interactions; due to these reasons, we feel that application classification is not a feasible approach.

DoS attacks are demonstrated in shared cache environment in which the data belonging to cache sensitive processes is evicted to force cache misses and slow them down [2, 11, 19, 35]. Intel CAT provides an attractive way of maintaining isolation through the use of CLOS groups as described in the next subsection. In our proposed work, we minimize sharing of cache-ways between reuse oriented processes, and in cases where must share, the duration of sharing is minimized thereby doing a best effort to minimize the possibility of DoS attacks. The framework coupled with the system assumptions in our opinion provides for a robust real world solution to the problem of cache based performance isolation. Our experimentation shows that the performance degradation
associated with full isolation in terms of cache ways is extremely severe ruling out such solutions. On the other hand, our best effort solution carefully allocates ways in terms of minimal sharing and thus the resulting performance degradation is negligible as shown in our empirical evaluation.

4.2 Intel® CAT: Overview & Terminology

Intel® Cache Allocation Technology (CAT) [10] allows the user to specify cache capacity designated to each running application. The primary objective of Intel CAT is to improve performance by prioritizing and isolating critical applications. Through isolation, Intel CAT also provides powerful means to stops certain kinds of side channel and DoS attacks. It provides a customized implementation of way-partitioning with a “software programmable” user-interface, that is used to invoke in-built libraries to perform cache partitioning.

To reason about different cache partitions, Intel CAT introduces the notion of Class-of-Service (CLOS), which are distinctive groups for cache allocation. The implication of the CLOS abstraction is that one or more applications belonging to the same CLOS will experience the same cache partition. Intel CAT framework allows users to specify the amount of cache allocated to each CLOS in terms of Boolean vector-like representations called Capacity Bitmasks (CBMs). These are used to specify the entire cache configuration for each CLOS, including isolation and overlapping of ways between one or more CLOS. On the hardware side, Intel CAT uses arrays of MSRs to keep track of CLOS-to-ways mappings. Minor changes are then done in the resource allocation mechanism of Linux Kernel to interact with these dedicated registers. In our work, we use apportioning algorithms to generate CBMs for a process residing in a particular CLOS. We then use specialized library calls to interact with the Intel CAT interface to perform the required allocation.

5 Probes Compiler Framework

We first describe the Compiler phase of probe insertion outlined in the overall system solution Fig. 2. Probes Compiler Framework is a LLVM-based instrumentation framework equipped with learning mechanisms to determine an application’s resource requirements across various execution points. It inserts ‘probes’ (specialized library markers) at each outermost nested-loop level within each function of an application. These probes estimate the attributes that directs an application’s cache requirements for an execution phase: memory footprint, cache sensitivity, reuse behaviour & phase timing.

In our work, we consider each nested loop to be an execution phase. Once inserted, the probes encapsulate these resource information and broadcasts them during execution to a proactive scheduler, which performs smart cache partitioning. While loop attributes like memory footprint and reuse behaviour can be statically analyzed and dynamically computed at runtime, other attributes such as phase timing have to be predicted using learning regression models which are embedded statically at compile time. In order to incorporate both the trained model & compute attributes dynamically, the Probes Framework has two components: compilation component and runtime component.

The compilation component primarily consists of multiple LLVM [15] compiler pass instruments probes into application and embeds loop memory footprint usage, data-reuse behaviour analysis and trained machine learning models for phase-timing. First, a preliminary pass profiles each loop nest to train regression models that predicts their execution times. The loop-phase timing is established as a linear function of the loop-bounds which constitutes the linear regression model. Apart from encapsulating loop attributes, the compilation component also inserts a special probe-start function in the preheader of the loop nest and a probe-completion function at the exit node of the loop nest. For loops present in the inner-nesting level, the probe functions are hoisted outside the outermost loop inter-procedurally. During hoisting, the pass combines all attributes of inner-loops with the outermost loop. For example, if any of the inner-most loop exhibits significant data reuse, then the entire loop-nest is considered to be a reuse-heavy phase. The runtime component on the other hand, compliments the compilation component by dynamically computing the values of memory footprint usage & phase-timing and conveying them to the proactive scheduler. This communication is facilitated by passing the attributes as arguments to the probe library function calls, which are further transferred to the scheduler via shared memory.

5.1 Phase Timing Model

The phase-timing is defined as the time taken for executing an entire loop-nest in an application and each loop-nest corresponds to an application phase. Probes Compiler Framework uses a linear regression model to predict the execution time for each loop-nest. The idea here is to establish a numerical relation between loop-timing and loop-iterations. Since total number of loop-iterations can be expressed in terms of the its bounds, the loop-time model can be further refined as a function of loop-bounds. In general, for a loop with arbitrary statements in its body, the loop-timing is proportional to the loop bounds. For nested-loops, bounds of each individual loops has to be included as well. To make this analysis easier, Probes Framework uses the LLVM [15] loop-simplify & loop-normalization passes, which transforms each loop to lower-bound of 0 and unit-step size. Thus, the timing model of a normalized loop-nest with n perfectly-nested loops having upper-bounds $u_1, u_2, ..., u_n$ is:

$$ T = f(u_1, u_2, ..., u_n), \quad (1) $$

To enable analysis of imperfectly nested-loops, loop-distribution can be performed to transform them into a series
of perfectly-nested loops. An example of semantically equivalent loops by loop distribution with same phase-time is shown in Fig. 4.

\[
\begin{align*}
\text{for (i = 0; i < u_1; ++i) } & \quad A[i] += A[i + 2] + i \\
\text{for (j = 0; j < u_2; ++j) } & \quad B[j] += B[j + 3] + j
\end{align*}
\]

Figure 4: Semantically equivalent loops by distribution

Therefore, the phase-timing equation can be decomposed sum of individual \( n \) distributed loops:

\[
T = f_1(u_1) + f_2(u_1, u_2) + ... + f_n(u_1, u_2, ..., u_n) \tag{2}
\]

Eq. 2 can be interpreted as a linear-regression model \( T_c(u) \) with weights \( c_1, c_2, ..., c_n \) & intercept \( c_0 \):

\[
T_c(u) = c_0 + c_1 u_1 + ... + c_n u_1 ... u_n \tag{3}
\]

Probes framework uses a specialized pass to generate loop-bounds & the timing for specific set of application inputs to generate test and training data for the regression model. Once the model is trained, the timing equation for each loop-nest is embedded in its corresponding probe. During runtime, the actual loop-bounds are plugged into this phase-timing equation to generate the phase-time, which is passed as an argument to the probe functions. For non-affine loops that have unknown bounds, we generate an approximate loop-bound based on the test input sets to predict the timing.

### 5.2 Loop Memory Footprint

**Memory footprint** is the amount of cache that will be utilized during an application phase. Probes framework calculates the memory footprint through static polyhedral analysis for the memory accesses in a loop nest. Polyhedral analysis creates memory access relations for each read and write access statements present in the loop body. These relations map the dynamic execution instances of the statements with the set of data elements accessed by them. Apart from this mapping, polyhedral access relations also contains the loop-invariant parameters (often unknown at compile time) and Presburger Formula, that captures the conditions surrounding the memory access. A simple example illustrating polyhedral access relation is depicted in Fig. 5.

In the above example, there are total four polyhedral access relations, two each for \( P \) & \( Q \) denoting the mapping of their dynamic instances with accessed elements. Since both the statements are enclosed within the same loop, the number of array elements of \( A \) & \( B \) accessed by these statements are same and is equal to \( N_s \). Now, the total number of unique data elements accessed in the array denotes memory footprint exhibited by the loop-nest. However, in this loop both statements have partially overlapping memory accesses over both arrays \( A \) & \( B \). Therefore, the total number of unique data elements can be found out by taking a union of memory elements accessed by both statements \( P \) & \( Q \) for arrays \( A \) & \( B \), denoted by \( n_A(P \cup Q) \) & \( n_B(P \cup Q) \):

\[
\begin{align*}
&n_A(P \cup Q) = n_A(P) + n_A(Q) - n_A(P \cap Q) = N + 7 \tag{4} \\
n_B(P \cup Q) = n_B(P) + n_B(Q) - n_B(P \cap Q) = N + 2 \tag{5}
\end{align*}
\]

where \( n_A(P) \) denotes the number of data elements of array \( A \) accessed by dynamic instances of \( P \) & \( n_A(P \cap Q) \) denotes the common elements of \( A \) accessed by \( P \) and so on. Thus, the total number of data elements accessed and the memory footprint of the loop nest is: \((2N + 9)\mid N \geq 0\). Therefore, at runtime with the value of loop upper-bound \( N \), probes calculates the exact footprint & passes it to the runtime component.

### 5.3 Classifying Reuse Behaviour

Most applications exhibit temporal and spatial locality across various loop iterations. To fully utilize the locality benefits, the blocks of memory with reuse potential has to remain in the cache, along with all the intermediate memory accesses. To determine the amount of cache required by a loop-nest to ensure that locality is maximized, we need to obtain a sense of reuse behaviour exhibited by the loop-nest. To classify reuse behaviour, Probes Framework uses **Static Reuse Distance (SRD)**, which is a measure of the number of memory instructions between two accesses to the same memory location. Based on this metric, the loop-nests could either be classified as **Streaming** - if its SRD is negligible, or else **Reuse** - if the SRD is significant and the reuse occurs over a large set of intermediate memory instructions. Consequently, reuse loops require significantly larger cache than streaming loop and this has to be accounted while deciding cache apportioning. Fig. 6 shows an example of how SRD can be leveraged to classify data-reuse behaviour of loop nests.

The loop-nest shown in the above example has statements \( S_1 \) & \( S_2 \), which have potential reuse of array \( A \) between them & statement \( S_3 \) has potential reuse of array \( B \) among iterations of outer-loop. Statement \( S_2 \) reads a memory location which is re-read by \( S_1 \) after two iterations of I-loop. However, since every iteration of I-loop comes with \( N \) intermediate iterations of statement \( S_3 \) from the inner-loop, the SRD between two
same accesses of \( A[i] \) from \( S_1 \) & \( S_2 \) will be \( 2 * N \). Similarly, each access to array \( B \) from \( S_3 \) is repeated after every iteration of outer-loop, which makes the SRD equal to \( M \). Since \( M >> N \), the SRD from \( S_1 \) & \( S_2 \) is insignificant and classified as streaming, while the SRD from \( S_3 \) classifies the inner-loop as reuse. Overall, the entire loop-nest is classified as reuse. At runtime, the exact SRD value is computed dynamically and passed on the probe functions. Similar to footprint estimation, approximate bounds are generated to compute SRD for loops whose bounds are unknown. Also, loops containing indirect subscripts or unanalyzable access patterns are classified as reuse since at runtime they could potentially exhibit a data reuse.

### 5.4 Analyzing Cache Sensitivity

Intuitively, the performance of a cache-sensitive application should improve with increase in allocated cache size. However, after an initial performance improvement most applications often exhibit a saturation point in their performance, which means that after a certain number of allocated cache-ways, there is no further performance benefit with more cache allocation. The number of cache ways that correspond to performance saturation point of an application, is defined as max ways. Probes Framework estimates this by executing the application with various cache sizes and plotting the execution time (or cache misses) corresponding to the allocated cache-ways. For applications that are cache-insensitive, the max-ways is assumed to be 2, as allocating less than that degrades performance by behaving as a directly-mapped cache.

Although max ways identifies whether an application exhibits cache-sensitive behaviour during its execution, it does not fully account for the degree of cache sensitivity. This is important because the more the cache-sensitivity an application has, the greater performance benefits it experiences with the same cache allocation. To quantify this varying degree of cache-allocation sensitivity among different applications, Probes Framework uses a metric called performance sensitivity factor (\( \alpha \)), that captures the change in an application’s loop nest execution times as a function of cache ways allocated to that application. For an application \( A \), the performance sensitivity factor can be defined as:

\[
\alpha_A = \sum_{i=3}^{\text{max ways}} \frac{|\Delta t_i|}{\Delta w_i}
\]

where \( \Delta t_i = |t_i - t_{i-1}| \) and \( \Delta w_i = |w_i - w_{i-1}| \). A higher value of \( \alpha \) indicates that the application’s performance benefits could be increased by changing the cache allocation when a particular loop nest is being executed. Probes profiles and sends the tuple \((\alpha, \text{max-ways})\) for an application to the scheduler at runtime to guide the apportioning algorithms.

### 6 BCache Allocation Framework

**BCache Allocation** is a compiler-guided cache apportioning framework that obtains efficient cache partitions for a diverse application mixes, based on their execution phases. The program execution phase is primarily characterized by the reuse behaviour of executing loop and its corresponding memory footprints, that are already predicted by the Probes Framework. Typically, during a particular instance of system execution, an application executing a reuse loop with a higher value of memory footprint is allocated a greater portion of the LLC, compared to another application executing a streaming loop. The phase change occurs whenever an executing loop changes its reuse-behaviour or exhibits a significant variation in its memory footprint & cache partitions are adjusted accordingly. The phase information is relayed by respective probes to the scheduler, which is further used by the apportioning algorithms to obtain phased-based cache partitions for co-executing applications. Also, the apportioning decisions of BCache minimizes shared-cache attacks by avoiding grouping of reuse-reuse process whenever possible.

An overview of BCache allocation framework’s working on a mix of \( N \) applications is shown in Fig. 7. These applications are instrumented by the Probes Framework. The overall BCache allocation framework is triggered upon application’s dynamic phase-changes & can be broadly categorized into two steps: Cache Apportioning based on per application demand and Cache Allocation Through CBM Generation to invoke Intel CAT and alter the cache configurations.

**Figure 7: BCache Allocation Framework**

BCache framework apportions the last-level cache among applications by adopting an unit-based fractional cache partitioning scheme. The idea here is that each application...
will be allocated a fraction of the LLC, which is measured by estimating how much they contribute to the entire memory footprint. The actual cache allocation is done in two parts - first, the cache demand for each application phase is calculated as a fraction of its memory footprints over all the other co-executing applications’ memory footprint. These footprints are adjusted to account for data-reuse nature of loop nests. The apportioning algorithms also makes use of other probes like phase-timing, max-way, etc to enhance the accuracy of partitioning decisions.

Secondly, based on the fractional cache amount calculated by the apportioning scheme for each running application, the framework then determines cache partitions in terms of capacity bitmasks (CBM) to invoke Intel® CAT. CBMs ensure that during application phase changes, the bitmasks are updated in a manner that is consistent with the overall system cache configuration. The other key factors in cache allocation include maintaining data locality and grouping compatible applications. Shifting an application from one cache partition to another during the course of its execution can jeopardize the benefits from data locality and increases compulsory misses. Therefore, we need to ensure that dynamically changing the cache allocations during application phase changes does not adversely affect data locality. Furthermore, sometimes when the number of running processes in a system increases or cache demand of application rises, sharing the LLC between multiple applications is inevitable. This necessitates grouping of more than one applications to share the same cache-ways and forcing two incompatible processes to be grouped together can adversely result in conflict misses. Therefore, determining application compatibility in cache-allocation is imperative. Using the notion of compatibility, the framework maximizes isolation of reuse sensitive loops in terms of the allocation of cache-ways to minimize the chances of DoS attacks.

Keeping these requirements in mind, BCACHE allocation framework uses two phase-aware cache partitioning algorithms: Initial Phased Cache Allocation (IPCA) and Phase Change Cache Allocation (PCCA). There algorithms generate CBMs in a manner that preserves data-locality and takes application compatibility into account. System socket selection and core allocation to applications are also taken into account while simultaneously limiting the number and the kind of loops/processes that can be grouped in the same CLOS to enhance security. IPCA and PCCA are invoked by the BCACHE scheduler for the initial cache allocations when the processes start and during phase change of each application respectively.

6.1 Cache Apportioning Scheme

BCACHE allocation framework determines cache apportions for an application based on loop memory footprint & data-reuse nature. First, the memory footprint of each loop is scaled as per whether it is a streaming loop or a reuse loop. Scaling ensures that the reuse loops get a bigger portion of cache than streaming loops. For an application \( K \) with current executing loop \( n \) having memory footprints \( m \), the adjusted loop memory footprint is defined as: \( km_{\text{phase}} = km * S_n \), where \( S_n \) denotes the scaling factor of the current loop \( n \) and can be adjusted dynamically. Depending on the reuse behaviour exhibited by a loop, the value of scaling factor ranges from: 
\[
S_n = \begin{cases} 
1, & \text{if } n \to \text{reuse} \\
(0, 1), & \text{if } n \to \text{stream}
\end{cases}
\]
Based on the adjusted footprint value and the adjusted footprints of the other executing loops in the system, a fraction of the cache size that will be allocated to the application is determined. This fraction is calculated by determining how much does the current application’s loop footprints contribute to the overall loop footprints in the system. In a system with \( N \) applications, the fraction of cache allocated to application \( K \) at time \( t \) is given by:

\[
f_{t}^{\text{cache}}(K) = \frac{km_{\text{phase}}}{\sum_{I=1}^{N} f_{t}^{\text{cache}}(I)}
\]

where \( km_{\text{phase}} \) denotes the adjusted loop memory footprint of application \( K \) and \( f_{t}^{\text{cache}}(I) \) denotes the adjusted loop memory footprint of application \( I \) in their current execution phases. At \( t = 0 \), all the applications will be allocated an initial fraction of cache according to the memory footprints of their first executing loop and the sum of these fractions for all the applications at \( t = 0 \) will be equal to 1. This means that unless applications are grouped in same CLOS, each running application will start their execution with an isolated portion of cache. However, an important observation in Eq. 7 is that each fraction is being calculated based on an individual application’s current phase and will be recomputed during their phase changes. Obviously, different co-executing applications in the system might not undergo phase change at the same time as others. This might result in an overlap between cache portions allocated to different applications and the sum of cache fractions will deviate from 1. Based on the sum of cache fractions for all applications in the system, Table 1 shows the three possible system scenarios at time \( t \).

| Sum of Cache fractions \( \sum_{I=1}^{N} f_{t}^{\text{cache}}(I) \) | System Scenario |
|-------------------------------------------------|-----------------|
| 1                                              | Cache is fully occupied and each application group has separate portion of cache |
| > 1                                            | Cache is fully occupied and some applications have overlapping cache portions |
| < 1                                            | Cache is not fully occupied and each application group have separate portion of cache |

Table 1: Three possible scenarios in fractional cache partitioning scheme
6.2 Phase-Aware Dynamic Cache Allocation

Generating cache partitions and their respective CBMs for each application from the cache fractions is achieved by Initial Phased Cache Allocation (IPCA) and Phase Change Cache Allocation (PCCA) algorithms. The IPCA algorithm (presented as Algo 1) is invoked whenever an application begins its execution. It is responsible for assigning a socket for each application. The sockets are selected based on the \( \alpha \) values and available cores. A separate socket is dedicated for applications with a higher value of \( \alpha \), while low- \( \alpha \) applications resides in the other sockets. If there are no cache-ways left in the high- \( \alpha \) socket, then applications are grouped in different sockets according to the available cores in the sockets.

Algorithm 1: Initial Phased Cache Allocation (IPCA)

Input: Process \( P \)
Result: Find effective cache partition for an application based on its initial phase

****** Initializing Socket ******
if \((P \rightarrow \alpha) > 1\) then
  if \(\text{high-} \alpha \text{Socket}\_\text{ways}() > (P \rightarrow \text{max ways})\) then
    Assign \(P\) to high- \( \alpha \) Socket
  end
else
  Assign \(P\) to a different socket with max available cores
end

****** Selecting CLOS ******
if \(\text{socket}\_\text{availCLOS} > (0.75 \times \text{socket}\_\text{totalCLOS})\) then
  find new clos for REUSE process
  find compatible clos for STREAM process
else
  group STREAM process in compatible clos having max(\(\Delta t\)) value
  group REUSE process in compatible clos having min(\(\Delta t/\alpha\)) ratio
end

****** Allocating cache ways ******
\(\text{req}\_\text{ways} = P \rightarrow \text{getCacheApportion}()\);
if \(\text{avail}\_\text{ways} \geq \text{req}\_\text{ways}\) then
  Allocate \(\text{req}\_\text{ways} \rightarrow P \rightarrow \text{CLOS}\)
else
  Allocate \(\text{avail}\_\text{ways} \rightarrow P \rightarrow \text{CLOS}\)
end
Generate appropriate bitmasks for allocated ways

After socket allocation, a suitable CLOS is obtained for the process. If less than 75\% of all CLOS groups on a socket are vacant, each reuse processes gets its own CLOS. For processes executing streaming loops, separate CLOS is assigned only if there is no compatible CLOS available in the socket. Consequently, when the number of occupied CLOS in a socket crosses the 75\% mark, both streaming and reuse loops are grouped in compatible CLOS groups. Next, the IPCA algorithm estimates the initial cache demand by using the fractional scheme, which is passed into a bitmask generator to obtain the required CBMs after checking with the available cache-ways in the system. In case the demand exceeds the available cache capacity, the framework allocates all possible ways to the CLOS and marks it as ‘unsatisfied’.

Finally, Intel® CAT is invoked by these generated bitmasks to create partitions.

The BCache scheduler obtains information broadcast from probes in the running applications. The probes convey to the scheduler of possible phase-changes in application and PCCA algorithm (presented as Algo. 2) is invoked to update the existing cache partitions based on the new requirements. The PCCA algorithm obtains the new demand by using the same fractional apportioning scheme and checks if the cache demand has changed or not. If demand increases, then extra ways are allocated to the CLOS if possible. However, if the current demand of the application can be satisfied with lesser ways, then extra ways are freed and allocated to the ‘most unsatisfied CLOS’ (\(\alpha_{\text{max}}\)) in the system. The required CBMs are obtained from the bitmask generator accordingly. A similar approach is followed when a loop finishes its execution, along with updating all system parameters like occupied CLOS, available ways, etc. Overall, the cache allocation algorithms are responsible for managing the following aspects of the system:

Algorithm 2: Phase-Change Cache Allocation (PCCA)

Input: Process* \(P\)
Result: Find efficient cache partition for an application based on its phase change

\(\text{req}\_\text{ways} = P \rightarrow \text{getCacheApportion}()\);
\(\text{curr}\_\text{ways} = P \rightarrow \text{ways}\);
if \(\text{req}\_\text{ways} > \text{curr}\_\text{ways}\) then
  \(\text{extra}\_\text{ways} = \text{req}\_\text{ways} - \text{curr}\_\text{ways}\);
  \(\text{avail}\_\text{ways} = \text{socket} \rightarrow \text{getavailWays}()\);
  if \(\text{avail}\_\text{ways} \geq \text{extra}\_\text{ways}\) then
    Allocate \text{extra}\_\text{ways} \rightarrow P \rightarrow \text{CLOS}
  else
    Allocate \text{avail}\_\text{ways} \rightarrow P \rightarrow \text{CLOS}
  end
else
  \(\text{free}\_\text{ways} = \text{curr}\_\text{ways} - \text{req}\_\text{ways}\);
  Allocate \text{free}\_\text{ways} to most unsatisfied \(P' \rightarrow \text{CLOS}\);
end

- **Preserving Data Locality**: All processes are confined to their initial socket throughout execution time and are not allowed to move to other socket which will jeopardize data locality. Applications that possess reuse loops with large footprints are kept within the same CLOS, to maintain a fixed subset of cache-ways for their entire execution period. On account of phase changes, the number of ways in that CLOS are expanded or shrunk in a way that ensures that an application doesn’t start over with entirely new cache.

- **Finding Compatible CLOS**: When total allocated ways reaches above 75\% of the maximum available ways, streaming processes are aggressively grouped together if their way-demands are similar and their difference in execution time is maximum. This allows preservation of ways for reuse processes. The compatibility relation for reuse processes involves similarity of their cache
demands in terms of ways needed, and if their \( \alpha \) differences and overlapping execution times \( \Delta t \) are minimum. This ensures that if all two or more reuse applications are grouped together if they require the same number of ways, their overall performance will not be impacted (\( \Delta_{\text{ways}} \)) and they won’t co-execute for long (\( \Delta_{\text{max}} \)).

- **Limiting Total Process in a CLOS**: The allocation algorithms makes sure that at any time instant, total processes in a given CLOS) \( \leq G\text{Factor} \) to avoid excessive thrashing.

### 7 Results & Evaluation

**Experimental Setup**: Com-CAS is evaluated on a Dell PowerEdge R440 server with supports way-partitioning through Intel CAT. Table 2 enlists our system configuration. The Probes Framework was compiled as an unified set of compiler passes in LLVM 3.8. For interacting with Intel CAT, pgos library 1.2.0 was used. For each socket in our system, one core and a single CLOS is kept vacant in order to avoid interfering with Daemon system processes.

| Machine          | Dell PowerEdge R440 |
|------------------|---------------------|
| Processor        | Intel Xeon Gold 5117 @ 2.00 GHz |
| OS               | Ubuntu 18.04, Linux Kernel 4.15 |
| Sockets          | 2 |
| Cores            | 16 per socket |
| L1 Cache         | 8-way set-associative, 32 KB private |
| L2 Cache         | 16-way set-associative, 1 MB private |
| L3 Cache         | 11-way set-associative, 19 MB shared |

Table 2: System Specification

**Benchmark Choice**: Three diverse sets of benchmarks are chosen for our experiments: PolyBench Suite [21], a numerical computational suite containing 30 individual benchmarks solving linear algebra problems and data analysis applications, GAP Benchmark Suite [1], a standard graph processing workload containing 7 benchmark kernels from problems in graph theory & Rodinia [5], which is a compute-intensive heterogeneous benchmark suite having 20 benchmarks from different domains like medical imaging and pattern recognition. From all these suites, we excluded benchmarks that had insignificant execution time since they don’t exhibit cache-sensitive behaviour & their execution time makes the results non-repeatable. Table 3 shows the final 36 benchmarks that were used that represent a variety of computational steps across a variety of domains that could execute in a multi-tenant environment.

**Creating Effective Workload Mixes**: To test the potential of our allocation algorithms, the mixes should be composed of processes that can potentially saturate the system in terms of either cache demand or core demand. Keeping this constraint in mind, a total of 35 mixes were created, with approximately 8 processes per mix on average. The applications with \( \alpha > 1 \) are added in every mix since we are interested in determining the effectiveness of our algorithms in sensitive benchmarks to improve throughput.

**Prediction Accuracy**: The loop timing, shown in Figure 10 had an average of 86% with a minimum of 64% and maximum of 100%. On the other hand, memory footprint, shown in Figure 11 had an average of 92.86% with a minimum of 84% and maximum of 100%. Both loop timing and memory footprint are affected by the hoisting of probes because it replaces exact values with expected values leading to less precise timing and footprint values. This result can be seen especially GAP benchmarks because these benchmarks tend to have many interprocedural loops compared to Polybench and Rodinia. For the benchmarks that are high in Polybench, these are the small benchmarks that did not change drastically from training inputs leading to very high accuracies. Even if these two variables might not be super accurate, they may not have a huge factor in an allocation if a majority of the ways are already allocated or the maxways of the loop cuts it off. Since the accuracy for both is above 85%, it provides reasonable allocations that improved execution time.

### 7.1 Improvement in Execution Time

Com-CAS’s performance on all 35 mixes from Polybench, GAP and Rodinia is summarized in Fig. 12. We compare Com-CAS with an unpartitioned cache, where the mixes are simply compiled with normal LLVM and executed on Linux’s CFS Scheduler [14], which is the most commonly used scheduler in real-world systems. The mix completion times were noted on the baseline system (unpartitioned cache) and Com-CAS
and speedup of completion time due to \textit{COM-CAS} over the base was calculated. \textit{Com-CAS} had an average improvement of \textbf{28.4\%} over all 15 Polybench mixes, \textbf{8\%} over all 10 GAPS mixes and \textbf{25.3\%} over all 10 Rodinia mixes. In general, \textit{Com-CAS} obtains completion or throughput speedup in every single application mixes over the system with unpartitioned cache.

In addition, \textit{Com-CAS}'s performance maintains SLA agreements of applications in all mixes. The SLA agreement, in our case, is performance degradation below 15\% of their ‘original-unmixed’ time. The original-unmixed time is measured by running the process individually in an unpartitioned cache system. To best show this SLA agreement, we select two representative mixes from each benchmarks shown in Figure 13: the mix that obtained maximum speedup (best-performing mix) over unpartitioned cache and mix that obtained minimum speedup (worst-performing mix). Overall, across all 271 individual processes distributed in 35 mixes, \textit{Com-CAS} achieves an average performance degradation of \textbf{1.13\%}, compared to their original-unmixed time, and they all were within the SLA agreement, i.e. none showed a degradation of worse than 15\%. In some cases, the applications do better than original and this can be due to code transformations done by our LLVM probe insertion pass.

### 7.2 Effect on Cache Misses

The allocation algorithms in BC\textit{Cache Framework} focuses enhancing the overall system performance by apportioning higher amount of cache to reuse-based processes that exhibit higher cache-sensitivity (high \textit{α}). As a result, the general trend observed is that reduction in LLC cache misses are shifted towards reuse-based applications that "need a greater amount of cache". This is important since the system throughput is determined by the execution time of process that has the longest execution time in mix and typically, they demand more cache and are cache-sensitive. The processes \textit{Floyd-Warshall} (Polybench), \textit{BC} (GAP) & \textit{SRad} (Rodinia) showed a reduction of \textbf{27.56\%}, \textbf{3.44\%} & \textbf{44.9\%} respectively. For other applications present in the mix, the cache misses are either the same or have somewhat increased. This is because BC\textit{Cache Framework} prioritizes applications which exhibits higher degrees of cache-sensitivity. Also, for non-cache sensitive applications that have been instrumented with probes exhibit more misses because of the additional probe functions. However, the resulting penalty is still within the 15\% SLA limit. Overall, \textit{Com-CAS} achieved a reduction of \textbf{4.6\%} in LLC misses over all the mixes.

### 7.3 Case Studies

We now take a closer look at particular mixes and show exactly how \textit{Com-CAS} affects the applications. We look at the worst (\textit{M22}) and best (\textit{M25}) performing GAP Mixes with their performance improvements shown in Figure 13.

Mix 22 consisted of \textit{BC}, \textit{CC}, and \textit{CC\_SV} processes, that are all cache sensitive. In the initial allocation, \textit{BC} was placed in Socket 1 with 2 ways, \textit{CC} was placed in Socket 1 with 2 ways, and \textit{CC\_SV} was placed in Socket 0 with 2 ways. All processes begin with 2 ways because the first loop nest, which is common between all benchmarks in GAP, is a streaming loop that only required 2 ways at max. The max ways do not change across the execution, so each application ends with 2 ways. \textit{CC} ends first followed by \textit{CC\_SV} then \textit{BC} with a \textbf{2\%} mix speedup and a \textbf{0.66\%} decrease of total LLC cache misses.

Mix 25 consisted of 2 \textit{BCs}, 2 \textit{CCs}, and 2 \textit{SSSPs}. Compared to \textit{CC\_SV}, \textit{SSSP} is considered cache insensitive. Both \textit{BCs} were placed in Socket 1 and both \textit{CCs} and \textit{SSSPs} were placed in Socket 0. Each \textit{CC} and \textit{BC} are placed in different CLOS starting with 2 separate ways. The two \textit{SSSPs} share the same CLOS meaning they share the same ways. Eventually, each \textit{CC} application will decrease his ways from two to one then increase it back to two. Just like the previous mix, each process ends with two ways. The order of completion is \textit{CC}, \textit{BC}, \textit{SSSP}, and \textit{SSSP}. Overall, the mix speedup is \textbf{18\%} and a \textbf{0.92\%} decrease of total LLC cache misses.

### 7.4 Allocation Metrics

Across all the mixes, our \textit{Com-CAS} did 203 allocations with 157 of them being initial apportions. During the executions of these mixes, \textit{Com-CAS} increased partitions 20 times and shrunk partitions 26 times. As the most allocations happened initially, our system finds it to be optimal for the mixes to not change ways a lot. In the cases that they did change, \textit{Com-CAS} increased partitions 20 times and shrunk partitions 26 times with The largest increase being 3 with decrease being 2. In total across all mixes, there were only 157 CLOS groups. Typically, the streaming applications can share CLOS groups with other applications. On the other hand, reuse oriented applications do not tend to be within the same CLOS group. Our \textit{Com-CAS} only had 11 CLOS groups containing more than one reuse oriented application leading to the probability of a CLOS group with at least 2 reuse applications being 7\%. This probability being low means the less likelihood of an application doing an LLC DoS attack on another application due to the reuse separation. In addition, a majority of the loops in reuse oriented applications are streaming meaning reuse loops do not happen at the same time. As a result, the
probability of LLC DoS attacks are lower than 7%. Moreover the timing overlap of loops sharing the CLOS was minimized while co-locating them in the same CLOS.

8 Conclusion
In this work, we propose a Compiler-Guided Cache Apportioning System (Com-CAS) for effectively apportioning the shared LLC leveraging Intel CAT under compiler guidance. Probes Compiler Framework evaluates cache-related loop properties such as loop property (streaming or reuse), cache footprint, loop timings saturation factor (in terms of number of ways), and cache sensitivity and relays them to apportioning scheduler. On the other hand, BCache Allocation Framework uses allocation algorithms to schedule processes to sockets, cores, and CLOS then dynamically partitions the cache based on the above information. Our system improved average throughput by 21%, with a maximum of 54% while maintaining the worst individual application execution time degradation within 15% to meet SLA requirements in a multi-tenancy setting. In addition, the system’s scheduling minimizes the co-location of reuse applications within the same CLOS group along with their overlap. With improved throughput, fulfilled SLA agreement, and increased security of processes, our Com-CAS is a viable system for multi-tenant setting.

References
[1] S. Beamer, K. Asanović, and D. Patterson, “The gap benchmark suite,” arXiv preprint arXiv:1508.03619, 2015.

[2] M. G. Bechtel and H. Yun, “Denial-of-service attacks on shared cache in multicore: Analysis and prevention,” CoRR, vol. abs/1903.01314, 2019. [Online]. Available: http://arxiv.org/abs/1903.01314
[3] D. J. Bernstein, “Cache-timing attacks on aes,” 2005.

[4] J. Chang and G. S. Sohi, “Cooperative cache partitioning for chip multiprocessors,” in ACM International Conference on Supercomputing 25th Anniversary Volume, 2007, pp. 402–412.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in 2009 IEEE international symposium on workload characterization (IISWC). IEEE, 2009, pp. 44–54.

[6] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, “A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness,” ACM SIGARCH Computer Architecture News, vol. 41, no. 3, pp. 308–319, 2013.

[7] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, “Kpart: A hybrid cache partitioning-sharing technique for commodity multicores,” in 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2018, pp. 104–117.

[8] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, “Polly-polyhedral optimization in llvm,” in Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT), vol. 2011, 2011, p. 1.

[9] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+ flush: a fast and stealthy cache attack,” in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2016, pp. 279–299.

[10] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, “Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family,” in 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016, pp. 657–668.

[11] S. Khan, G. Mruru, and S. Pande, “A compiler assisted scheduler for detecting and mitigating cache-based side channel attacks,” arXiv preprint arXiv:2003.03850, 2020.

[12] S. Kim, D. Chandra, and Y. Solihin, “Fair cache sharing and partitioning in a chip multiprocessor architecture,” in Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004. IEEE, 2004, pp. 111–122.

[13] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, “Dawg: A defense against cache timing attacks in speculative execution processors,” in 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Oct 2018, pp. 974–987.

[14] J. Kobus and R. Szklarski, “Completely fair scheduler and its tuning,” draft on Internet, 2009.

[15] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, 2004, pp. 75–86.

[16] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, “Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems,” in 2008 IEEE 14th International Symposium on High Performance Computer Architecture. IEEE, 2008, pp. 367–378.

[17] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in 2015 IEEE Symposium on Security and Privacy, May 2015, pp. 605–622.

[18] F. Mueller, “Compiler support for software-based cache partitioning,” ACM Sigplan Notices, vol. 30, no. 11, pp. 125–133, 1995.

[19] N. Pimpalkar and J. Abraham, “A llc-based dos attack technique on virtualization system with detection and prevention model,” in 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2018, pp. 419–424.

[20] L. Pons, J. Sahuquillo, V. Selfa, S. Petit, and J. Pons, “Phase-aware cache partitioning to target both turnaround time and system performance,” IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 11, pp. 2556–2568, 2020.

[21] L.-N. Pouchet et al., “Polybench: The polyhedral benchmark suite,” URL: http://www. cs. ucla. edu/pouchet/software/polybench, 2012.

[22] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’06). IEEE, 2006, pp. 423–432.

[23] R. Ravindran, M. Chu, and S. Mahlke, “Compiler-managed partitioned data caches for low power,” ACM SIGPLAN Notices, vol. 42, no. 7, pp. 237–247, 2007.
[24] D. Sanchez and C. Kozyrakis, “The zcache: Decoupling ways and associativity,” in 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2010, pp. 187–198.

[25] ——, “Vantage: scalable and efficient fine-grain cache partitioning,” in Proceedings of the 38th annual international symposium on Computer architecture, 2011, pp. 57–68.

[26] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, “Application clustering policies to address system fairness with intel’s cache allocation technology,” in 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, 2017, pp. 194–205.

[27] T. Sherwood, B. Calder, and J. Emer, “Reducing cache misses using hardware and software page placement,” in Proceedings of the 13th international conference on Supercomputing, 1999, pp. 155–164.

[28] G. E. Suh, L. Rudolph, and S. Devadas, “Dynamic partitioning of shared cache memory,” The Journal of Supercomputing, vol. 28, no. 1, pp. 7–26, 2004.

[29] D. Tam, R. Azimi, L. Soares, and M. Stumm, “Managing shared l2 caches on multicore systems in software,” in Workshop on the Interaction between Operating Systems and Computer Architecture, 2007, pp. 26–33.

[30] R. Wang and L. Chen, “Futility scaling: High-associativity cache partitioning,” in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2014, pp. 356–367.

[31] X. Wang, S. Chen, J. Setter, and J. F. Martínez, “Swap: Effective fine-grain management of shared last-level caches with minimum hardware support,” in 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2017, pp. 121–132.

[32] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, “Secdcp: secure dynamic cache partitioning for efficient timing channel protection,” in Proceedings of the 53rd Annual Design Automation Conference, 2016, pp. 1–6.

[33] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, l3 cache side-channel attack,” in 23rd USENIX Security Symposium (USENIX Security 14). San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[34] Y. Ye, R. West, Z. Cheng, and Y. Li, “Coloris: a dynamic cache partitioning system using page coloring,” in 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT). IEEE, 2014, pp. 381–392.

[35] T. Zhang, Y. Zhang, and R. B. Lee, “Dos attacks on your memory in cloud,” in Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ser. ASIA CCS ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 253–265. [Online]. Available: https://doi.org/10.1145/3052973.3052978

[36] X. Zhang and Q. Zhu, “Hica: Hierarchical cache partitioning for low-tail-latency qos over emergent-security enabled multicore data centers networks,” in ICC 2020-2020 IEEE International Conference on Communications (ICC). IEEE, 2020, pp. 1–6.

[37] X. Zhang, S. Dwarkadas, and K. Shen, “Towards practical page coloring-based multicore cache management,” in Proceedings of the 4th ACM European conference on Computer systems, 2009, pp. 89–102.

[38] H. Zhu and M. Erez, “Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems,” in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, 2016, pp. 33–47.