FLORIA: A Fast and Featherlight Approach for Predicting Cache Performance

Jun Xiao
University of Amsterdam, Netherlands
Peking University, China
sunny-xiaojun@hotmail.com

Yaocheng Xiang
Peking University, China
yaocheng_x@pku.edu.cn

Xiaolin Wang
Peking University, China
wxl@pku.edu.cn

Yingwei Luo
Peking University, China
lyw@pku.edu.cn

Andy D. Pimentel
University of Amsterdam, Netherlands
A.D.Pimentel@uva.nl

Zhenlin Wang
Michigan Technological University
USA
zlwang@mtu.edu

ABSTRACT

The cache Miss Ratio Curve (MRC) serves a variety of purposes such as cache partitioning, application profiling and code tuning. In this work, we propose a new metric, called *cache miss distribution*, that describes cache miss behavior over cache sets, for predicting cache MRCs. Based on this metric, we present FLORIA, a software-based, online approach that approximates cache MRCs on commodity systems. By polluting a tunable number of cache lines in some selected cache sets using our designed microbenchmark, the cache miss distribution for the target workload is obtained via hardware performance counters with the support of precise event based sampling (PEBS). A model is developed to predict the MRC of the target workload based on its cache miss distribution.

We evaluate FLORIA for systems consisting of a single application as well as a wide range of different workload mixes. Compared with the state-of-the-art approaches in predicting online MRCs, FLORIA achieves the highest average accuracy of 97.29% with negligible overhead. It also allows fast and accurate estimation of online MRC within 5*ms*, 20*X* faster than the state-of-the-art approaches. We also demonstrate that FLORIA can be applied to guiding cache partitioning for multiprogrammed workloads, helping to improve overall system performance.

CCS CONCEPTS

• **General and reference** → **Performance**; **Metrics**; **Measurement**.

KEYWORDS

Shared Cache, Performance Prediction, Cache management, Cache Performance, Locality Modeling

ACM Reference Format:
Jun Xiao, Yaocheng Xiang, Xiaolin Wang, Yingwei Luo, Andy D. Pimentel, and Zhenlin Wang. 2023. FLORIA: A Fast and Featherlight Approach for Predicting Cache Performance. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3577193.3593740

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.
ICS '23, June 21–23, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0056-9/23/06…$15.00
https://doi.org/10.1145/3577193.3593740

1 INTRODUCTION

Modern multicore processors implement a large Last Level Cache (LLC) to hide the long memory access latencies. Such an LLC is usually shared by multiple cores to allow high cache utilization and to provide convenient communication among cores. However, an uncontrolled shared cache allows CPU cores to freely access the entire cache space, which can cause inter-application cache interference when multiple applications compete among each other for the shared LLC. This can increase cache misses for each individual application and consequently degrade the overall system performance.

Even though the size of the LLC is constantly increasing, it is still one of the most critical resources that needs to be managed well in order to reach the performance potential of the memory hierarchy. The Miss Ratio Curve (MRC), a performance-directed metric, was proposed for the purpose of improving LLC management. It identifies the cache miss rate of an application as a function of the amount of cache allocated to that application, i.e., its cache occupancy.

In general, the MRC provides deep insights into the locality characteristics of a program and serves as a popular tool to enable various different types of analyses: memory management prediction [4, 5], cache simulation [25], profiling and code tuning [20, 27] and so on [2, 7, 28]. Online cache MRCs can also help to guide the cache partitioning for multiprogrammed workloads running on a multicore processor with shared LLC to mitigate the shared cache contention [13, 33, 34, 36, 42, 44].

A relatively straightforward way to obtain MRCs is to do it *offline* by running the target application multiple times, each time using a different cache size. However, in a real system, especially in data centers, it may be impractical to profile every workload in advance. In addition, an offline MRC might be inaccurate or even useless as it is input dependent.

Accurate MRCs can be calculated by measuring stack distance [22, 26, 29, 35, 36]. Stack distance is the number of distinct accesses between two consecutive accesses to the same location. To reduce the time and space complexity of calculating stack distance, recent works, such as StatCache [4], StatStack [12], footprint theory [41], time-to-locality conversion [17, 31, 38] and AET model [15, 42] use reuse interval [11] to construct MRCs more efficiently. Reuse time

simply counts the total number of accesses between two consecutive accesses to the same data location. However, those approaches based on stack distance and reuse interval are limited by the following disadvantages: (1) They need the full history of memory access traces, which requires either exhaustive binary instrumentation or interrupting the thread on every memory access. Even with program phase-based sampling, these approaches incur substantial slowdowns, of 21% to over 2× [30], making them too slow for online purposes. (2) Those approaches target only caches using relatively simple replacement policies like least-recently used (LRU) or pseudo-LRU. LLCs in modern processors, however, employ high-performance replacement policies [40] to improve their caching performance over conventional policies like LRU.

Contributions: We make three main contributions in this work. First, we propose a new metric that describes cache miss behavior over cache sets, called *cache miss distribution*, for MRC prediction. This metric can be measured by tracking the cache misses of a program over a time interval. When accessing a memory address results in an LLC miss, we count one cache miss for the LLC set to which the address will be mapped. This metric leverages one key observation: cache misses are normally distributed over all the cache sets. It intrinsically differs from stack distance and reuse interval in terms of four aspects: (1) As the cache miss behavior over all cache sets can be represented by the behavior of a group of cache sets, it only monitors a small group of *individual cache sets*, instead of the *whole* cache space, (2) It only counts the number of misses over some cache sets, without distinction of data addresses for calculating stack or reuse distance, (3) it does not require recording every cache miss, it is able to maintain a high MRC prediction accuracy even when sampling cache misses at low rates, (4) Since it is not inherently tied to any particular cache replacement policy, it is applicable to modern LLCs.

Second, based on the proposed metric, we present the design and practical implementation of FLORIA, a software-based, online approach that approximates cache MRCs on commodity systems with low overhead. More specifically, a microbenchmark, called CACHEBUBBLE, is designed and implemented to pollute the cache at the granularity of cache lines, forming different thrashing patterns at cache sets. By adjusting CACHEBUBBLE's access pressure on some selected cache sets, the number of cache lines in those sets that are available to the target application also changes. FLORIA uniquely utilizes hardware performance counters with the support of precise event based sampling (PEBS) to obtain the cache miss distribution for the target workload. A model is developed to predict the MRC of the target workload based on its cache miss distribution.

Third, we evaluate FLORIA for systems consisting of a single application and a wide range of different workload mixes. Compared with the state-of-the-art, FLORIA achieves the highest average accuracy of 97.29%, while it incurs negligible overhead. It also allows fast and accurate estimation of online MRC within 5*ms*, 20*X* faster than the state-of-the-art approaches. This is particularly useful in the serverless computing where the execution time of workloads can be as low as hundreds of milliseconds [18]. In these cases, FLORIA allows to explore optimization opportunities for resource scheduling and management at millisecond timescales. We performed a sensitivity study for FLORIA and also demonstrate that FLORIA can be applied to guiding cache partitioning for multiprogrammed workloads, helping to improve overall system performance.

The rest of the paper is organized as follows. The background of hardware performance monitoring units and LLC is introduced in Section 2. Section 3 presents the observation on cache miss distribution for a target workload. Section 4 describes FLORIA, where we also detail the design of the microbenchmark CACHEBUBBLE and the model for MRC prediction. Section 5 presents the performance evaluation of FLORIA. Section 6 gives an overview of related work, after which Section 7 concludes the paper.

2 BACKGROUND

In this section, we introduce the background knowledge on hardware performance monitoring units and LLC.

2.1 Hardware PMUs

To provide real-time microarchitectural information about the processes currently executed on the chip, a rich set of hardware Performance Monitoring Units (PMUs) are implemented in today's processor microarchitectures. These PMUs offer a programmable way to count hardware events such as CPU cycles, instructions executed, cache statistics, etc. PMUs also support advanced event sampling, a mechanism that collects event samples at a predefined sampling period. For example, the event-based sampling is realized by Intel's Precise Event-Based Sampling (PEBS) [8] and AMD's Instruction Based Sampling (IBS) [10].

2.2 Last Level Cache (LLC)

Caches in modern processors are organized as a hierarchy of multiple cache levels to address the tradeoff between cache latency and hit rate. The low level caches are usually private to cores, while the last level caches (LLC) are shared between all cores.

The LLC consists of cache sets with a minimum unit of a cache line or cache block. An *M-way set-associative* cache allows a memory address to map to one of M cache lines in a set, from way 1 to way M. When a CPU needs to access a specific memory address, it checks whether a cache line containing the target address exists or not. If such a cache line is found, a cache hit occurs. Otherwise, it results in a miss which may incur a cache replacement. The cache replacement policy determines which block in a set is evicted for the new data. LLCs typically follow an approximation of the least recently used (LRU) policy for replacement.

LLC addressing. The LLC in a modern multicore processor is usually organized into as many slices as the number of cores with the purpose of reducing the bandwidth bottleneck when more than one core attempts to retrieve data from the LLC at the same time. All slices are addressable and can be accessed by all cores as a single logical LLC. Modern processors map a physical address to a slice in the LLC using an undocumented technique called *complex addressing*.

We consider an M-way set-associative LLC with a total of K cache sets in each cache slice. A cache line with a size of C bytes occupies a single way of a cache set. As LLCs are physically indexed and physically tagged (PIPT), they use the physical address for both the index and the tag. The slice and cache set to which a physical

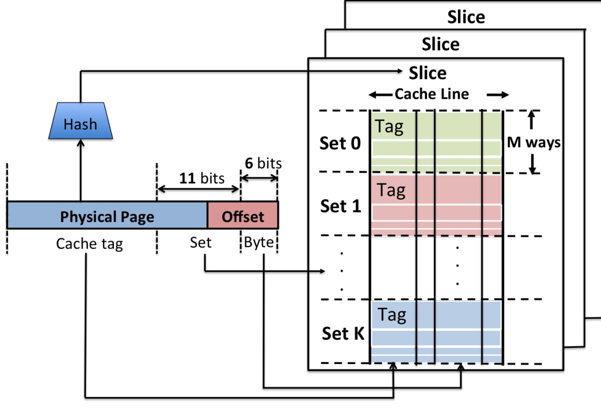

Figure 1: LLC addressing. For a typical LLC with $K = 2048$, $C = 64$, the lowest 6 bits ($b_0 - b_5$) are used to determine the offset within a cache line and $b_6 - b_{16}$ select the cache set. Higher bits b_{17} and above) are used as tag and input to a hash function to decide the cache slice.

memory address maps is determined by its address bits, as shown in Figure 1.

As indicated in [16], the least significant $\log_2 C$ bits of the physical address are used to address a byte or word within a cache line. The next $\log_2 K$ bits select the set that the cache line belongs to. Bits $\log_2 K \log_2 C$ and above are utilized as a tag for comparison when looking for data in the cache. The hash function also takes these higher bits as input and its output determines the cache slice.

Reverse Engineering of Complex Addressing. There have been many efforts to find the undisclosed hash function that determines the mapping between physical addresses and slices [16, 19, 23]. In this work, we adopt the approach presented in [23] to perform the reverse-engineering for the processor used in our experiments (Intel Xeon Silver 4110). In Intel processors, each LLC slice is equipped with a C-Box counter. C-Box can be configured to measure hardware events for its associated slice such as the total number of lookups or misses. The approach is applicable to any processor that is equipped with uncore performance monitoring units e.g., C-Box counters.

In more detail, this approach involves two steps:

Step 1: mapping between physical addresses and LLC slices. The C-Box counters are configured to count all accesses to each slice. Next, a specific virtual address is repeatedly accessed $10,000$ times to generate access events on the corresponding slice. All C-Box counters are then read for each slice. The virtual address is then translated to a physical address by reading the page tables in the file `/proc/self/pagemap`. Finally, a C-Box counter that has the most lookups will identify the slice to which that particular physical address is mapped.

By applying the same technique to different addresses, we can obtain a set of pairs (physical address, slice) that, eventually, form a mapping table.

Step 2: constructing the hash function. As validated in [23], the LLC hash function of an Intel CPU with 2^n cores can be expressed as a series of XORs of the bits of the physical address. This allows us to analyze the implication of the address bits independently from each other and reduce the analysis to only a handful subset of physical addresses. Specifically, one can compare the slices found by the previous step for different physical addresses that only differ by one bit. If the two addresses are mapped to the same slice, it means that the bit is not part of the hash function. Conversely, if the mapped slices are different, the bit is one of the inputs of the hash function. By performing the above analysis to each bit in a physical address, the hash function can be constructed.

We let b_i $0 \leq i \leq 63$ denote bit i of a 64-bit address. The number of LLC slices in our tested machine is 8, thus the hash function has an output of 3 ($\log_2 8$) bits. For simplicity, we express the hash function as three boolean functions o_s, $0 \leq s \leq 2$, each determines one bit of the output. Let I_s be the set of bits that are used to calculate o_s, i.e. $b_i \in I_s$ means bit i is one of the input bits to generate o_s. After performing the step 1 and 2, we found I_s for our test machine as follows, which is the same as the one identified by [23]:

$$\begin{aligned} I_0 &= \{b_6, b_{10}, b_{12}, b_{14}, b_{16-18}, b_{20}, b_{22}, b_{24-28}, b_{30}, b_{32-33}, b_{35-36}\}, \\ I_1 &= \{b_7, b_{11}, b_{13}, b_{15}, b_{17}, b_{19-24}, b_{26}, b_{28-29}, b_{31}, b_{33-35}, b_{37}\}, \\ I_2 &= \{b_8, b_{12-13}, b_{16}, b_{19}, b_{22-23}, b_{26-27}, b_{30-31}, b_{34-37}\}. \end{aligned} \tag{1}$$

Therefore, $o_s 0 \leq s \leq 2$ can be calculated by:

$$o_s = \oplus b_i, b_i \in I_s. \tag{2}$$

In general, reverse engineering of complex addressing may not be always feasible. However, the security community was able to recover the mapping for a wide range of platforms including Intel's Ivy Bridge, Nehalem and Haswell families with Intel Xeon, i5, i7 processor and so on.

In Section 4.1, we will exploit the hash function to generate addresses mapped to different cache slices in the microbenchmark, CACHEBUBBLE.

LLC partitioning. To provide hardware support for LLC partitioning, Intel has proposed the so-called Cache Allocation Technology (CAT), which provides software-programmable control over the amount of cache space that can be consumed by a given application.

Machines that support CAT have a predefined number of classes of service (CLOS), for example, 11 in our experimental machine. Each CLOS is associated with a capacity bit mask (CBM) that controls the accessibility of cache resources with cache-way granularity, where each bit in the mask grants write access to one way in the cache. Each application belongs to a CLOS and a particular application can only access the cache-ways defined by the CBM for that CLOS.

3 CACHE MISS DISTRIBUTION

Different from previous work that exploits either stack distance or reuse time, FLORIA relies on the new metric – cache miss distribution – for predicting an MRC.

When accessing a memory address results in an LLC access, it then checks whether the LLC set to which the address is mapped contains the target address or not. If not, we count one cache miss for that particular LLC set. We use a cache miss histogram to represent the distribution of LLC misses over all the LLC sets.

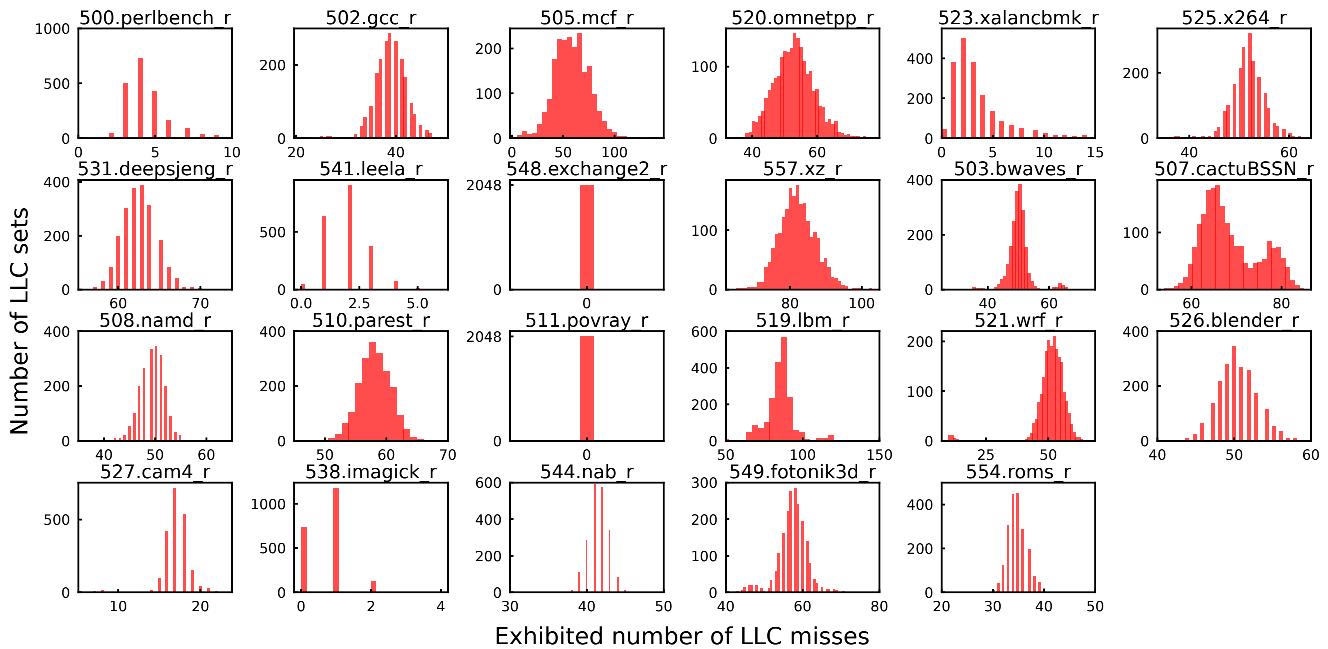

Figure 2: LLC miss histograms with respect to cache sets.

The cache miss histogram describes cache miss behavior over cache sets. It can be obtained by the following steps: virtual addresses of the LLC misses can be obtained by using the PMU sampling mechanism. The tracked virtual addresses can be translated to the corresponding physical addresses. Given a physical address, one can determine its associated LLC slice and the cache set within the slice where the address is mapped to. By sampling the LLC misses over a short execution period, one can obtain the cache miss distribution over cache sets. We describe those steps in detail below.

PMU sampling. Intel PEBS is an event-based sampling mechanism that allows associating sampled performance events with instruction pointers (IP) and effective data addresses. PEBS address sampling in recent Intel processors (i.e., Haswell and its successors) allows precisely monitoring cache misses. As we are interested in LLC misses, we choose the event `MEM_LOAD_UOPS_RETIRED:L3_MISS` to record addresses whose access results in an LLC miss.

Virtual-to-physical address translation. This translation is done via `Pagemap`, a set of interfaces in the Linux kernel that allow user space programs to examine the page tables and related information.

Since the default page size of most Linux systems in the virtual address space is 4K bytes, during the virtual-to-physical address translation, bits $0-11$ ($b_0 - b_{11}$) of the virtual address that encode the page offset are preserved. Bits 12 and above of the virtual address, which encode the page number in the virtual address space, are replaced by the physical page frame number. The mapping from the virtual page to the physical page frame can be found in `/proc/self/pagemap`, a component in `Pagemap`.

LLC addressing. As described in Section 2.2, given a physical address whose access results in an LLC miss, one can determine the associated LLC slice and the cache set within the slice where the address is mapped to.

We performed the above steps to obtain the cache miss behavior for each applications in the SPEC CPU2017 benchmark suite [32] when hardware prefetchers are disabled. We calculate the average number misses at each set across all cache slices, based on which we count the number of LLC sets that exhibited the same number of misses. Figure 2 illustrates such LLC miss histograms with respect to cache sets for each application in the SPEC CPU2017 benchmark suite.

In Figure 2, for a biomedical imaging application (*510.parest*), all cache sets exhibit between 51 and 66 misses, with an average of 57.65. 97.12% of all sets experience a number of misses that ranges from 52 to 63. Only less than 3% of cache sets exhibit misses out of 10% of the average value. We can also observe similar cache miss distribution for the most programs in the SPECCPU2017. Note that there are a few exceptions: (1) for some programs such as *548.exchange2*, *511.povray* and *538.imagick*, they have very limited LLC misses, (2) for most programs, a few cache sets may exhibit more/less cache misses than the median value. This can be due to software prefetch, compiler optimizations such as alignment and so on.

Observation 1. *Cache misses are normally distributed over all the cache sets. The cache miss behavior at some randomly selected cache sets coincides with the behavior of all cache sets together.*

As shown in Figure 1, bits 6-16 of an address select the cache set. Given a large number of data addresses that lead to misses in the LLC, we observe that bits 6-16 of those addresses are either 0 or 1 with the same probability. Therefore, a missed data address

has a uniform probability of being mapped to any of the LLC sets. Observation 1 also verifies the assumption that a program block has the same probability of being present in any of the cache sets in the work on analytic cache models [1].

The cache miss behavior over all cache sets can be represented by the behavior of some individual cache sets. Instead of monitoring the whole cache space, we can focus on the miss behavior of some individual cache sets. By exploiting Observation 1, FLORIA creates different degrees of contention on selected cache sets and analyzes the cache miss behavior at those cache sets to produce an MRC.

4 FLORIA: DESIGN AND IMPLEMENTATION

In this section, we describe our approach, FLORIA, for predicting cache performance. An overview of FLORIA is presented in Figure 3. FLORIA relies on a microbenchmark, CACHEBUBBLE, to create contention on the shared cache with the target application. The role of CACHEBUBBLE is to pollute the LLC sets at the granularity of cache lines, forming different thrashing patterns at some selected cache sets. The LLC access behavior of CACHEBUBBLE can be controlled in such a way that CACHEBUBBLE can access a certain number of cache lines in a specific cache set within each cache slice. By adjusting the number of cache lines accessed by CACHEBUBBLE in different cache sets, the available cache lines in those cache sets for the target application will differ. While the target application executes concurrently with CACHEBUBBLE, its cache miss distribution over the controlled cache sets is obtained by performing the steps described in Section 3. Finally, the MRC of the target application can be predicted using its cache miss behavior. In the following, we explain the approach in detail. We start by introducing the CACHEBUBBLE microbenchmark.

4.1 The CACHEBUBBLE Microbenchmark

CACHEBUBBLE acts as a cache set polluter. It first generates data addresses and then frequently accesses them.

The procedure of CACHEBUBBLE is shown in Pseudocode 1. The inputs to CACHEBUBBLE include the number of cache slices S, the number of cache ways M, the execution *duration* of CACHEBUBBLE, the set of cache sets it will access: $SampleSet = c_1, c_2, \ldots$, and the number of cache lines to access for each set in $SampleSet$: $W = w_{c_1}, w_{c_2}, \ldots$.

Supposing the total number of cache sets in $SampleSet$ is N_{sample}, we set an offset $x \in 0, \left\lfloor \frac{K}{N_{sample}} \right\rfloor - 1$ for selecting one cache set among every $\left\lfloor \frac{K}{N_{sample}} \right\rfloor$ cache sets to form $SampleSet$. This cache set selection facilitates designing an address filter, as will be explained in Section 4.2.

We first allocate a buffer in a huge page of 1GB by using `mmap`, which is contiguous physical memory aligned on the page size (Line 2 in Pseudocode 1). The offset of a memory block in a 1GB page is 30-bit long, so the lowest 30 bits of a virtual address within a huge page will be the same as those bits in the corresponding physical address. Based on this property, we can generate addresses within a huge page that map to a specific LLC set by only setting the lower bits of virtual memory addresses without knowing precisely the actual physical addresses. Note that we have no means to manipulate bit 30 and above in the physical address in user space, as it is controlled by physical page allocation in the operating system.

Pseudocode 1: CACHEBUBBLE

```
1:  Input: S, M, duration, SampleSet, W
2:  buffer ← create_1G_hugepage()
3:  for all c ∈ SampledSet do
4:      addr ← buffer + (c«6)
5:      φ_c[S] ← gen_addr_all_slices(addr)
6:      for i ← 0 to S-1 do
7:          ψ_{c,i} ← gen_access_addr(φ_c i, M)
8:      end for
9:  end for
10: start ← read current time stamp
11: while end − start ≤ duration do
12:     for iteration ← 0 to 20 do
13:         for c ∈ SampledSet do
14:             for i ← 0 to S − 1 do
15:                 access the first w_c addresses in ψ_{c,i}
16:             end for
17:         end for
18:     end for
19:     end ← read current time stamp
20: end while
```

4.1.1 *Generating Addresses.*

To create LLC access pressure, each data access of CACHEBUBBLE is expected to result in a L1/L2 cache miss and a LLC hit. In order to guarantee that CACHEBUBBLE behaves in this way, we first need to carefully generate those data addresses.

Address mapped to a given cache set. We first construct a set of addresses that map to the given cache set c. As shown in Figure 1, bits $b_6 - b_{16}$ determine the cache set and they are the same for both virtual and physical addresses within the allocated huge page. By setting $b_6 - b_{16}$ to c, we can generate such an address $addr$ that maps to the set c (Line 4 in Pseudocode 1).

Address mapped to different cache slices. To ensure that CACHEBUBBLE creates the same amount of access pressure on a given cache set at each cache slice, we need to distinguish the cache slice an address maps to. In this way, we can expect that each memory access of CACHEBUBBLE will result in an LLC hit, avoiding L1/L2 cache hits or LLC misses due to the unbalanced distribution of accesses to cache slices.

Supposing that address $addr$ maps to cache slice with index $o_2 o_1 o_{02}$, we now generate a set of addresses that map to set c but in other cache slices by changing those bits of an address that determine the cache slice, as shown in Equations 1 in Section 2.2.

The mapped cache slice is calculated by Equation 2. Note that the output of o_s is determined by the number of input bits that equal to 1 in I_s. If the number of $1'$s in I_s is odd, the output of o_s is 1. If it is even, then o_s is 0.

If one bit above b_{16} in I_s is flipped (from 0 to 1 or vice versa), the number of 1's in I_s will change either from odd to even or from even to odd. Consequently, the value of o_s is also flipped, generating a new address that maps to a different slice. For example, flipping b_{17}

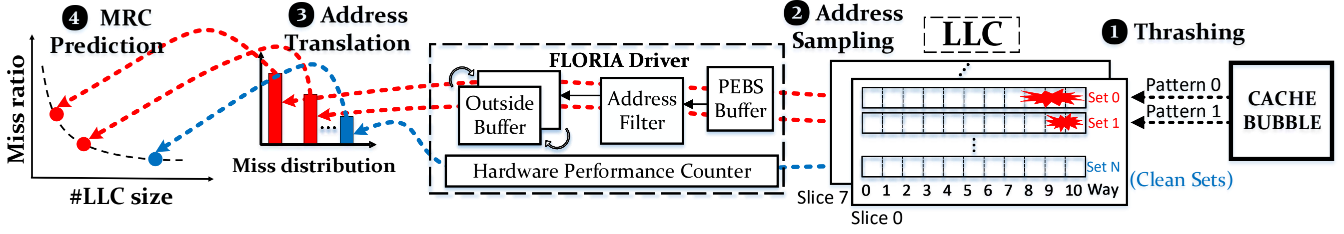

Figure 3: Overview of FLORIA.

of *addr* generates an address mapped to cache set c in cache slice $o_2 o_1 \bar{o}_{02}$, and flipping b_{21} produces an address mapped to cache slice $o_2 \bar{o}_1 o_{02}$.

By flipping bits in I_s, CACHEBUBBLE generates addresses that map to every cache slice. This step is done by the `gen_addr_all_slices` function at Line 5 in Pseudocode 1. It returns an array $\phi_c S$ that stores S addresses, one per cache slice.

Addresses mapped to the same cache set and slice

The generation of addresses that are mapped to the same cache set and slice follows from the observation that flipping an even number of bits in I_s will not change the value of o_s.

For example, flipping both b_{24} and b_{28} of an address generates a new address mapped to the same cache set and slice. CACHEBUBBLE generates a total number of M addresses mapped to cache set c in cache slice i, as performed by `gen_access_addr` at line 7 in Pseudocode 1. Those addresses form the set $\psi_{c,i}$.

4.1.2 Accessing Addresses. After address generation, CACHEBUBBLE frequently accesses those addresses and exploits the cache replacement policy to create different degrees of access pressure on the selected cache sets. If CACHEBUBBLE is configured to occupy w_c ($w_c \leq M$) cache lines in set c of slice i, then CACHEBUBBLE accesses the first w_c addresses in $\psi_{c,i}$. The accessing activity is performed by the while loop in Pseudocode 1 for a certain period, given by *duration*.

Thrashing pattern. The set *SampleSet* is further divided into M subgroups, each of which consists of L cache sets. CACHEBUBBLE accesses the same amount of cache lines (ways) for each cache set in the same subgroup, forming a thrashing pattern. In total, CACHEBUBBLE creates M thrashing patterns, one per subgroup, by accessing different numbers of cache lines (ways) ranging from 0 to $M-1$ for the M subgroups.

CACHEBUBBLE creates contention on totally L cache sets for each thrashing pattern to reduce the measurement error. However, with a larger L, CACHEBUBBLE evicts more cache sets per thrashing pattern, which leads to performance degradation for the target workload. In this work, we choose $L=4$ as the trade-off between prediction accuracy and application performance degradation.

4.2 Cache Miss Behavior

While the target workload and CACHEBUBBLE execute simultaneously on different processing cores, we monitor the cache miss behavior for the target workload.

We implemented a kernel driver, called `FLORIA Driver`, to collect runtime execution information for the target workload. We use the `MEM_LOAD_UOPS_RETIRED:L3_MISS` event counter with a predefined *sampling period* to drive the PEBS sampling for the target workload. When the predefined number of LLC misses (length of a sampling period) occurs, a PEBS record containing the linear address of a memory reference that triggers the current LLC miss is written into the configured PEBS buffer. When the PEBS buffer is full, it triggers an interrupt. In the interrupt handling function, linear addresses in the PEBS buffer are passed through a designed address filter and then are dumped to an outside buffer through `mmap`. For each sampled address in the outside buffer, virtual-to-physical address translation and LLC addressing are performed to obtain the cache miss behavior for the target workload, of which the details are described in Section 3.

The address filter is designed to reduce the size of the buffer storing the sampled virtual addresses and to also reduce the overhead incurred by unnecessary virtual-to-physical address translations. This is motivated by the fact that PEBS samples those virtual addresses that result in a LLC miss, no matter in which cache set the miss occurs. However, we are only interested in cache misses in those cache sets polluted by CACHEBUBBLE with different thrashing patterns.

The design of the address filter. By default, the virtual page assigned to the target application is 4KB, within which the lower bits $b_0 - b_{11}$ of a virtual address are the same as those bits in the corresponding physical address. As bits $b_6 - b_{16}$ of a physical address determine its mapped LLC cache set, bits $b_6 - b_{11}$ of the data address sampled by PEBS can be used to filter out uninteresting addresses.

Remember that bits $b_6 - b_{16}$ of a physical address determine its mapped LLC cache set and that the offset for selecting cache sets to be accessed by CACHEBUBBLE is x (see Section 4.1). When an address is sampled by PEBS, min6, $\lceil \log_2 \frac{K}{N_{sample}} \rceil$ bits starting from b_6 of that address are extracted. Only if the extracted value equals to x, the address will be recorded for address translation later.

For example, if CACHEBUBBLE accesses totally 64 cache sets, each cache set with an offset 10 among every 32 cache sets is selected to form *SampleSet*. Only the addresses, sampled from the target workload, with $b_6 - b_{10}$ that equal to 10 will be recorded. Thanks to the address filter, only 132 of virtual addresses captured by PEBS will be collected, which reduces both the buffer size and the overhead of address translation by about 97%.

4.3 MRC Prediction

We now develop a model for the prediction of an MRC based on the cache miss distribution.

As shown in Section 3, if the target application executes alone, cache misses are normally distributed over all cache sets, thus the

number of cache misses exhibited by each cache set is very similar. However, if the target application co-runs with CACHEBUBBLE, the cache miss distribution is not normal anymore. The target application exhibits more misses in the cache sets where CACHEBUBBLE thrashes more cache lines. This is because if more lines in a specific cache set are polluted by CACHEBUBBLE, fewer cache lines in that cache set are effectively available to the target application.

We obtain the cache miss distribution of *502.gcc* from the SPEC CPU2017 suite when running it concurrently with CACHEBUBBLE. According to CACHEBUBBLE's design, it thrashes certain (ranging from 0 to $M-1$) cache lines for M subgroups of cache sets.

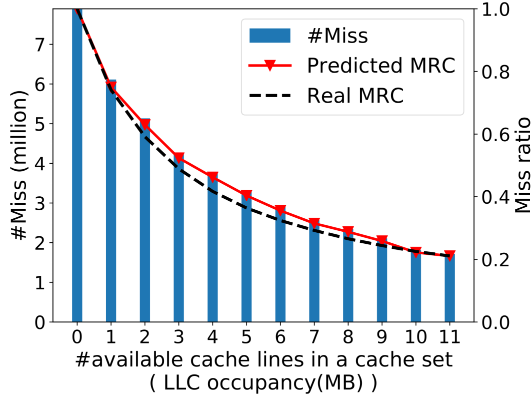

Figure 4: The comparison between normalized cache miss in the sampled cache sets with different numbers of available cache lines and real MRC for *502.gcc*.

We first calculate the average number for cache misses for each subgroup of cache sets with i cache lines effectively available for the target application, i.e., CACHEBUBBLE thrashes $M-i$ cache lines in those sets. The average numbers of cache misses for each subgroup are then normalized.

We compare the normalized cache miss numbers with the real MRC[1]. The comparison is depicted in Figure 4, from which we make the following observation:

OBSERVATION 2. *The normalized cache miss numbers from the cache sets with i cache lines effectively available for the target application are proportional to the cache miss ratio when an $i-way$ set associative cache is utilized by the target application.*

We denote $\hat{MR}_i$ as the predicted cache miss ratio of the target application when an $i-way$ set associative cache is utilized by the target application. Let $N_i 0 \leq i \leq M$ be the average number of misses in the cache sets where i cache lines are available for the target application.

Following Observation 2, we need to know at least one pair of $N_i, \hat{MR}_i$ to construct the MRC. Fortunately, two pairs can be derived. The first pair is $N_0, 100\%$, simply comes from the fact that cache miss ratio is 100% if no cache is available for the target application . The second pair is N_M, MR_c, where MR_c is the current cache miss ratio. MR_c can be measured by perf [9] at run time. As CACHEBUBBLE only pollutes a small subset of all cache sets ($N_{sample} << L$), it has very limited influence on the current cache miss ratio of the target workload. Therefore, MR_c is the cache miss ratio when the target application fully utilizes cache.

Using the above two pairs, $\hat{MR}_i$ is computed by:

$$\hat{MR}_i = \begin{cases} \frac{N_i}{N_M} \times MR_c, & \text{if } \frac{N_i}{N_M} \leq \frac{N_0}{N_i}, \\ \frac{N_i}{N_0} \times 100\%, & \text{otherwise.} \end{cases}$$

4.4 Limitation of Applicability

In general, FLORIA can be applied to an architecture if (i) the mapping between physical addresses and cache sets/slices is known, and (ii) the advanced event-sampling mechanism is supported. For the first condition, normally the mapping implemented by Intel, AMD and ARM is undocumented, but one can perform reverse engineering to discover the mapping for a wide range of their processor. Cache architectures in RISC-V are configurable, thus it is difficult to find a general approach to discover the mapping for RISC-V. However, if a RISC-V processor is open sourced, the mapping can be derived from the implementation. For the second condition, Intel processors provides PEBS, and AMD with IBS. ARM-based processors and RISC-V have no direct implementation of event sampling, but they can support it by adding related hardware performance counters.

5 EXPERIMENTS

This section evaluates the performance of our approach. The experimental platform is an Intel Xeon Silver 4110 @2.1 GHZ. The L1 size is 32K, L2 size is 1MB and the LLC size is 11MB. It has 376GB of main memory and the maximum memory bandwidth is 119.21 GB/s, so the memory contention will be small. Hyperthreading is disabled to avoid intra-core interference. By default, hardware prefetching is also disabled and the PEBS sampling period is 1.

5.1 Accuracy

We use applications from the SPEC CPU2017 benchmark suite [32]. As FLORIA is designed for online MRC prediction, we evaluate its performance using a fixed-work methodology [14]. For each application, it first executes for about 10 seconds to take over the cache, which is considered as a warm up interval. The MRC of the next 1 second execution is then predicted by FLORIA, OPMRC [42] and DynaWay [13].

We first predict the MRC when the target workload runs alone in the system. Figure 5 compares the real MRC and the MRCs produced by OPMRC [42], DynaWay [13] and FLORIA for each application in the SPEC CPU2017 benchmark suite.

The accuracy for a single application is calculated by $\frac{1}{M}{}_{i=1}^{M} | \hat{MR}_i - MR_i |$, where MR_i is the real cache miss ratio when an $i-way$ cache is allocated to the application using CAT.

The comparison of MRC prediction accuracy between FLORIA and other approaches is listed in Table 1. Overall, FLORIA achieves the best accuracy with an average of 97.29%, while the average accuracy obtained by DynaWay and OPMRC are 92.63% and 94.43%, respectively. For some programs (*557.xz, 519.lbm and 544.nab*), the improvement of FLORIA is small, as DynaWay and OPMRC can already predict MRCs with a 99% accuracy. For other programs, the improvement is larger. For *507.cactuBSSN*, the accuracy of FLORIA

[1]The real MRC is obtained offline by running the target application multiple times with the available LLC space controlled by CAT.

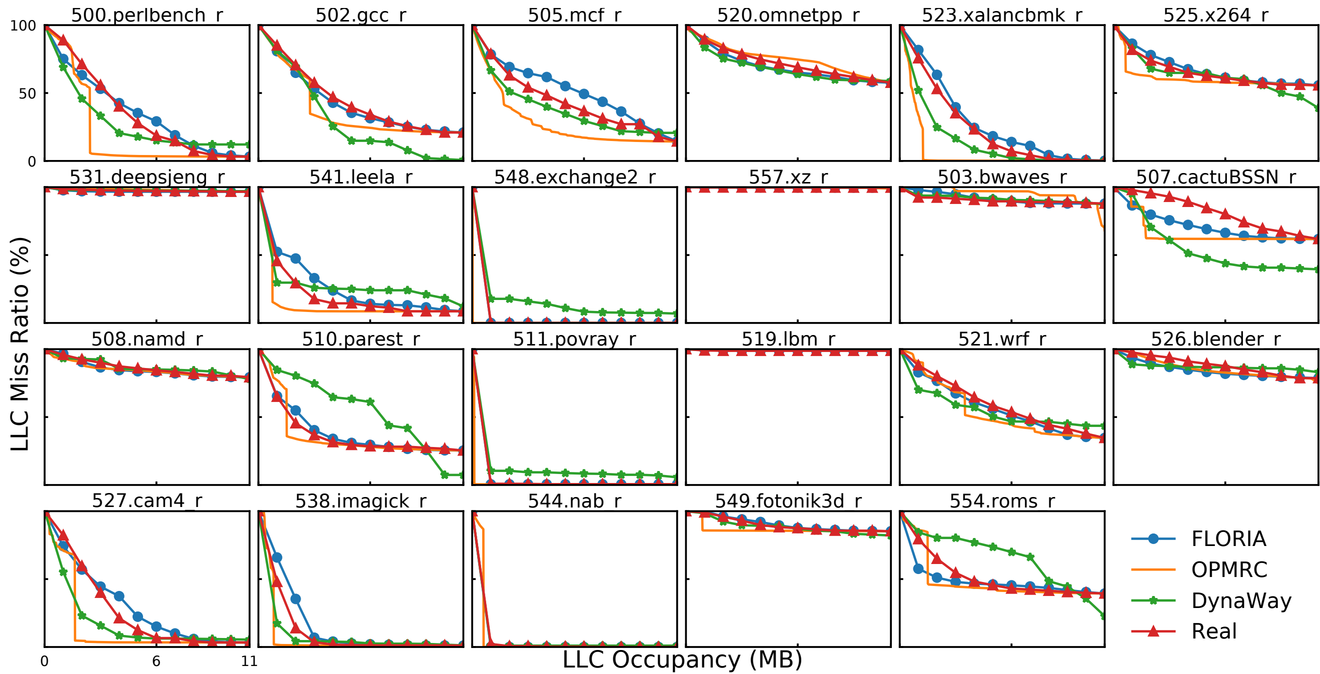

Figure 5: The LLC miss ratio (%) over cache size (MB) predicted by FLORIA, OPMRC and DynaWay, compared with real MRC.

is 90.46%, while the accuracy of DynaWay and OPMRC is 74.56% and 85.17%. In the best case, FLORIA is 11.05% more accurate than OPMRC (for *523.xalancbmk*) and 19.73% more accurate than DynaWay (for *510.parest*).

Table 1: Accuracy of FLORIA, DynaWay and OPMRC

Benchmark	FLORIA	DynaWay	OPMRC
500.perlbench	95.39 %	88.95 %	86.83 %
502.gcc	97.74 %	85.47 %	92.81 %
505.mcf	93.27 %	93.21 %	85.96 %
520.omnetpp	97.39 %	95.95 %	97.35 %
523.xalancbmk	96.22 %	91.45 %	85.17 %
525.x264	98.45 %	95.99 %	95.47 %
531.deepsjeng	99.50 %	99.68 %	99.48 %
541.leela	94.67 %	90.71 %	93.41 %
548.exchange2	99.99 %	89.86 %	99.99 %
557.xz	99.93 %	99.92 %	99.67 %
503.bwaves	98.42 %	98.90 %	88.18 %
507.cactuBSSN	90.46 %	74.56 %	85.60 %
508.namd	98.66 %	98.57 %	98.68 %
510.parest	98.09 %	78.33 %	96.97 %
511.povray	99.85 %	92.89 %	99.85 %
519.lbm	99.89 %	99.67 %	99.88 %
521.wrf	97.16 %	92.55 %	94.52 %
526.blender	96.45 %	95.83 %	96.78 %
527.cam4	95.35 %	90.37 %	88.58 %
538.imagick	96.01 %	95.75 %	94.71 %
544.nab	99.64 %	99.01 %	99.64 %
549.fotonik3d	99.44 %	98.38 %	96.52 %
554.roms	95.57 %	84.53 %	95.89 %
Avg.	97.29 %	92.63 %	94.43 %

FLORIA is more accurate than DynaWay because it predicts the entire MRC in one time using the obtained cache miss behavior at controlled cache sets. Compared with FLORIA, DynaWay requires multiple measurements, one per each evenly spaced allocation (1, 3,..., 11 ways), to construct the entire MRC. The application can behave differently in each measurements. Using the cache miss ratios measured at different phases causes MRC prediction errors.

OPMRC is less accurate than FLORIA for two reasons. First, OPMRC relies on the reuse time metric for MRC prediction, which is designed for caches with LRU replacement policy. However, the processor used for the experiments does not use LRU (or pseudo-LRU) for the LLC. Second, OPMRC requires a sampling frequency of 1, but the actual sampling frequency supported by commodity processors usually cannot reach this ideal value. With the tested processor, we observe a maximum sampling rate of 1:3.5. This inevitable factor introduces some inaccuracy for OPMRC.

5.2 Overhead

The actual run time overhead incurred by the online MRC prediction approaches depends on the frequency of phase transitions and duration of the measured execution phase. We first determine the change of an execution phase by 10% variation of the cache miss ratio. The average lengths of an execution phase for each application are listed in Table 2. Due to space limitations, the applications are represented by their indexes.

For each application, we choose an execution phase that lasts longer than 1 seconds. FLORIA and other approaches do not need to process the prediction during the whole phase. Instead, they only measure part of the execution phase and then use the predicted MRC at the measured interval as the MRC of the whole phase.

Length of Measurement Window. By varying the measurement lengths, FLORIA, DynaWay and OPMRC are used to predict the MRCs of the chosen execution phase for each application. Figure 6 summaries the average accuracy of those approaches.

Table 2: Overhead of FLORIA, DynaWay and OPMRC

Bench	Phase length (Second)	Application slow down[2]		Overhead		
		CAT	PEBS	F	OP	Dyna
500	110.17	5.69%	1.32%	0.00%	0.00%	0.01%
502	4.11	11.93%	29.58%	0.04%	0.72%	0.29%
505	3.92	6.06%	55.22%	0.07%	1.41%	0.15%
520	609.16	4.69%	92.18%	0.00%	0.02%	0.00%
523	47.73	28.81%	8.86%	0.00%	0.02%	0.06%
525	19.58	0.71%	22.50%	0.01%	0.11%	0.00%
531	14.69	0.57%	12.40%	0.00%	0.08%	0.00%
541	38.79	0.00%	0.36%	0.00%	0.00%	0.00%
548	814.19	0.40%	0.48%	0.00%	0.00%	0.00%
557	11.75	9.28%	31.93%	0.01%	0.27%	0.08%
503	1.84	1.03%	83.33%	0.23%	4.54%	0.06%
507	0.66	5.22%	55.88%	0.42%	8.47%	0.79%
508	1.03	0.44%	25.95%	0.13%	2.52%	0.04%
510	2.03	9.77%	22.74%	0.06%	1.12%	0.48%
511	659.04	0.39%	1.06%	0.00%	0.00%	0.00%
519	371.01	4.11%	106.35%	0.00%	0.03%	0.00%
521	1.51	6.19%	37.84%	0.13%	2.51%	0.41%
526	0.84	3.01%	17.89%	0.11%	2.13%	0.36%
527	8.39	1.88%	21.54%	0.01%	0.26%	0.02%
538	37.55	0.30%	0.62%	0.00%	0.00%	0.00%
544	7.34	0.00%	17.02%	0.01%	0.23%	0.00%
549	29.32	2.94%	184.21%	0.03%	0.63%	0.01%
554	1.83	5.74%	131.65%	0.36%	7.18%	0.31%
Avg.	121.59	4.75%	41.78%	0.07%	1.40%	0.19%

Given the requirement that the prediction accuracy shall be higher than 90%, the measurement length of FLORIA is 5ms, while 100ms for OPMRC and DynaWay. The MRCs prediction of FLORIA is $20X$ faster than OPMRC and DynaWay, given the same degree of accuracy. This feature makes FLORIA suitable for predicting the MRCs of both programs with frequent phase changes and small jobs that execute for less than $20ms$.

Table 2 lists the application slowdown caused by the three approaches and the overhead in an execution phase with average length. During the prediction process of FLORIA and OPMRC, the overhead mainly comes from PEBS sampling. As FLORIA and OPMRC adopt the same sampling rate of 1, the application slowdown caused by PEBS sampling is same for FLORIA and OPMRC, which is averagely 41.78%. Note that during the measurement window of FLORIA, CACHEBUBBLE runs for $5ms$ on another core, resulting in an extra overhead of 0.004% in the execution phase, which is negligible.

In order to obtain the reuse time histogram that represents the locality of target application, OPMRC has to sample and analyze a sufficiently longer trace than FLORIA. The average overhead of OPMRC during the execution phase is 1.40%, which is 20× higher than FLORIA.

[2]Only occurs in the measurement window

DynaWay profiles multiple times to construct the entire MRC for a 11-way LLC. Each time CAT is applied to change the cache size that can be used by the application. The target application takes time to fill and leverage the new cache allocation. Profiling with reduced cache sizes also contributes to the slow down of the target application. On average, the application slow down is 4.75% while DynaWay is executing and the average overhead of DynaWay in the execution phase is 0.19%.

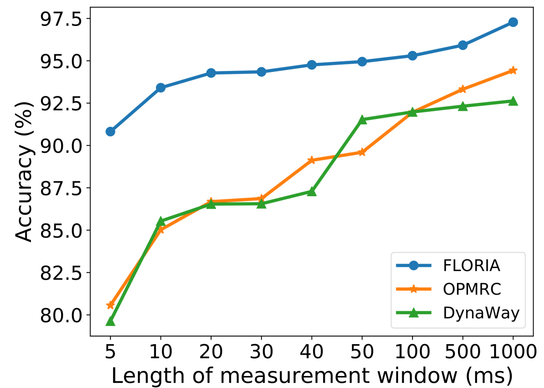

Figure 6: Average accuracy of FLORIA, DynaWay and OPMRC measured at different lengths of measurement window.

5.2.1 Co-run MRCs. We also predict the MRCs of multiple co-scheduled applications using FLORIA. Specifically, FLORIA profiles one application at a time. At each time, it splits cache capacity into two partitions: one with an $M-1$ way allocation for the target application and the other with a 1-way allocation shared by all remaining applications.

In each experiment, we randomly select 8 applications from SPEC to form a workload mix. In those experiments, FLORIA achieves the best average accuracy of 96.99%, while the accuracy of DynaWay and OPMRC is 89.11% and 92.31%, respectively. We find that the MRC prediction accuracy for each application in the workload mixes is very similar to the ones measured in the solo-run case. The overhead of FLORIA, DynaWay and OPMRC scales linearly with the number of MRCs predicted for the mix.

5.3 Sensitivity Analysis

The design of FLORIA involves trade-offs in accuracy and efficiency. In order to find the optimal parameters, we perform sensitivity studies in terms of different sampling periods. We also study the impact of hardware prefetching and memory bandwidth.

5.3.1 Effect of Sampling period. We perform experiments to compare the accuracy and overhead of FLORIA when PEBS is configured at different sampling periods. Figure 7 compares the accuracy of MRC prediction for each application when the PEBS sampling period is set to 1, 10, and 100. As can be seen, FLORIA is able to achieve a high average accuracy of more than 96% when the sampling period is 100.

For most applications, a higher prediction accuracy can be achieved if PEBS samples at a lower period. The reason is that with a lower

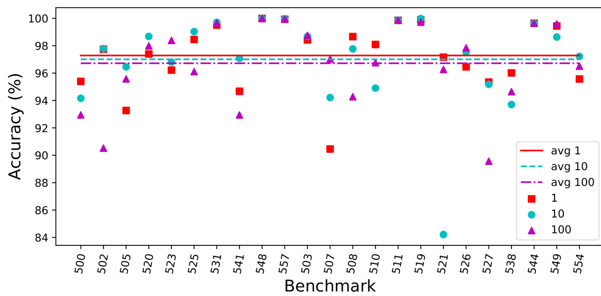

Figure 7: MRC prediction with different sampling periods.

sampling period, more cache misses at each cache set can be captured. Taking 508.*namd* as an example, Figure 8 compares the number of sampled misses with different sampling periods. When the sampling period increases from 1 to 100, the average number of misses per sampled set drops from 320 to 12, which leads to a larger measurement error.

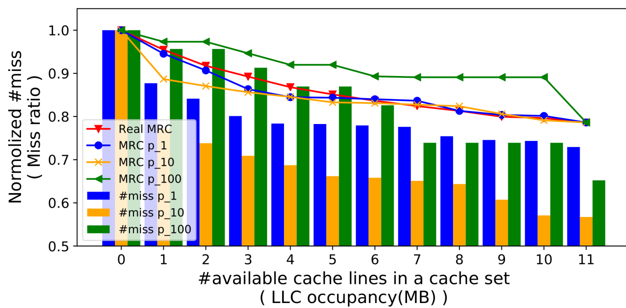

Figure 8: Breaking down the impact of sampling period on *508.namd*.

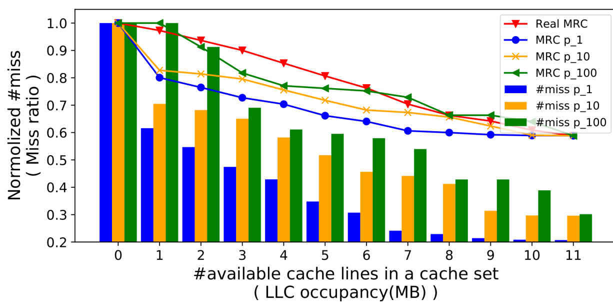

Figure 9: Breaking down the impact of sampling period on *507.cactuBSSN*.

Sometimes, accuracy increases with a larger sampling period, for example, 507.*cactuBSSN*. To investigate the reason, we show the cache miss distribution when sampling cache misses at different rates, as depicted in Figure 9. As can be seen, with a lower sampling rate, PEBS captures more misses per cache set and it is interesting to notice that PEBS samples more misses in the sets with more cache lines polluted by CACHEBUBBLE. This is because Intel precise event-based sampling (PEBS) can suffer from shadow effects [6, 24]: PEBS tends to capture memory accesses with a long latency in the pipeline and cache sets that exhibited more misses are sampled with a larger probability. As a result, the predicted cache miss ratio is lower than the actual one.

With the sampling period increasing from 1 to 100, the slowdown for the target workload caused by PEBS sampling decreases from 41.78% to 2.59% on average, while FLORIA is running.

5.3.2 The Impact of Hardware Prefetching. Hardware prefetchers, located in the L1/L2 caches, can have an impact on the real MRC. We perform experiments to investigate the accuracy of FLORIA when hardware prefetching is enabled.

With prefetching enabled, FLORIA achieves an average accuracy of 95.13%, which indicates hardware prefetching has very little influence on FLORIA. However, a few applications such as *500.perlbench* and *519.lbm* experienced about 10% accuracy loss. This is because hardware prefetching can affect the cache miss behavior at cache sets, which is observed in [43]. When hardware prefetching is enabled, the cache miss distribution obtained from PEBS sampling makes MRC prediction less accurate.

5.3.3 The Impact of Memory Bandwidth. As each data access of CACHEBUBBLE is designed to result in an LLC hit, CACHEBUBBLE does not access the main memory. Therefore, the performance of cache contention created by CACHEBUBBLE is not affected by the main memory bandwidth. We verified this by adopting the memory bandwidth allocation technique supported by Intel processors, to control the available memory bandwidth to CACHEBUBBLE. We found that FLORIA does not experience accuracy loss in predicting MRCs with a wide range of available memory bandwidths.

5.4 FLORIA for guiding Cache Partitioning

We briefly evaluate the usefulness of FLORIA when deploying it for guiding cache partitioning for multiprogrammed workloads.

Based on the MRC prediction of FLORIA, the workloads are divided into two groups: LLC-polluters and LLC-sensitive programs. In this experiment, we adopt a simple cache partitioning policy that allocates a small, 1-way partition to the group of LLC-polluters while letting the group of LLC-sensitive programs share the rest of the cache ways. We leave more complicated partitioning strategies as our future work.

We use an example workload that consists 8 concurrently executing programs selected from three different applications: $3 \times$ *519.lbm*, $3 \times$ *523.xalancbmk*, and $2 \times$ *510.parest*. At runtime, FLORIA predicts the MRC for each program. According to the prediction, the three instances of *519.lbm* are considered as LLC-polluters while *523.xalancbmk* and *510.parest* are classified as LLC-sensitive. Then, the LLC partitioning is applied. Figure 10 shows the LLC size occupied by each program without and with cache partitioning. Compared with the default case where all programs share the whole LLC, the IPC of *523.xalancbmk* and *510.parest* is increased by 11.4% (from 0.309 to 0.344) and 14.3% (from 0.851 to 0.973), respectively. At the same time, the IPC of *519.lbm* is reduced by only 0.5% (from

0.472 to 0.470). Overall, the system performance is improved by 7.75%.

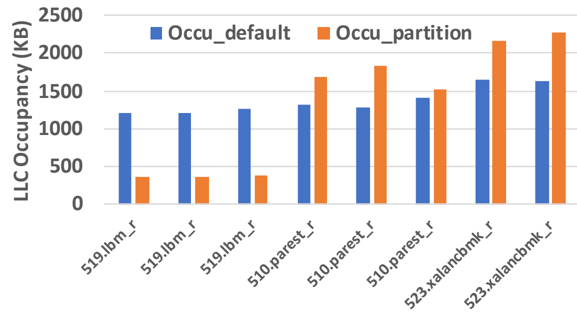

Figure 10: LLC occupancy of each program with and without cache partitioning

6 RELATED WORK

Mars et al. [21] designed a benchmark suite called SmashBench to quantify the sensitivity of a workload to cache and memory interference. SmashBench creates tunable resource contention by accessing the specified amount of caches and main memory. In comparison, our microbenchmark, CACHEBUBBLE, creates cache contention at the granularity of cache lines within the selected cache sets and its memory access behavior is fully controllable.

Accurate MRCs can be calculated by measuring stack distance [22]. Qureshi et al. [26] and Suhet et al. [35] proposed hardware access counters to record every cache access/hit to cache sets to track the stack distance. However, those performance monitoring counters are not available in commodity systems. FLORIA is different from UCP [26]: UCP first randomly selects cache sets to sample. Then it relies on monitoring units to record every cache hit to the selected cache sets.The hits count for each cache way to infer MRCs. FLORIA, on the other hand, first determines the cache sets to monitor together with the micro-benchmark CACHEBUBBLE. After that, PEBS is used to sample (i.e., not necessarily to record all) the misses at those chosen cache sets. We simply count the number of cache misses at the monitored cache sets and obtain the metric cache miss distribution. Counter Stacks [39] that uses probabilistic counters and SHARDS [37] that uses a splay tree are recent breakthroughs to reduce the cost of computing stack distance in practice. However, they target storage workloads and hence, cannot be directly applied to construct cache MRCs.

To reduce the time and space complexity of computing stack distance, recent studies [17, 31, 38, 41] use reuse time to construct MRCs more efficiently. StatCache [4] and StatStack [12] use counters to record reuse times for a particular set of references, these counters are then aggregated to form a reuse time distribution. Based on reuse times, Beckmann et al. [3] proposed a single framework consisting of hit, evict and age distributions to model caches with LRU and several recent policies for constructing cache MRCs. Hu et al. [15] proposed the AET model that monitors a fixed number of addresses for updating the reuse time histogram, which is then used to estimate the reuse time distribution for MRC prediction. However, the measurement of both stack distance and reuse time needs the full history of memory access traces, which requires either exhaustive binary instrumentation or interrupting the thread on every memory access. Even with program phase-based sampling, these approaches incur substantial slowdowns, of 21% to over 2× [30], making them too slow for online purposes.

A few tools have been developed to obtain the cache MRCs online. Tam et al. [36] proposed RapidMRC that uses IBM POWER5's specific SDAR performance counters for approximating L2 MRCs. Xiang et al. [42] designed OPMRC to obtain an online MRC. OPMRC first collects the LLC access trace on the fly and then constructs an MRC for the trace using the AET model. El-Sayed et al. [13] proposed DynaWay to construct MRCs by online profiling. By allocating different cache sizes to the target application using CAT, DynaWay periodically uses cache performance counters to infer the application's MRC.

FLORIA, presented in this paper, is fundamentally different from the above works in three aspects: (1) Instead of monitoring the whole cache space, it first determines a group of cache sets to monitor together with the micro-benchmark CACHEBUBBLE. (2) After that, it samples, rather than recording every miss at those selected cache sets via PEBS. (3) Instead of using metrics such as stack distance and reuse interval to approximate an MRC, FLORIA relies on the proposed metric, cache miss distribution, for MRC prediction.

7 CONCLUSION

In this work, we first proposed a new metric, *cache miss distribution*, that describes cache miss behavior over cache sets. Based on this metric, we presented the design and implementation of FLORIA, a software-based, online approach that approximates cache MRCs on commodity systems. FLORIA relies on CACHEBUBBLE to create cache pollution at the granularity of cache lines. When running CACHEBUBBLE together with the target application, FLORIA exploits hardware features of performance monitoring units with the support of PEBS to obtain the cache miss distribution for the target workload. We evaluated FLORIA for systems consisting of a single application as well as for a wide range of workload mixes. Compared with the state-of-the-art approaches in predicting online MRCs, FLORIA achieves the highest average accuracy of 97.29% with negligible overhead. It also allows fast and accurate estimation of online MRC within $5ms$, $20X$ faster than the state-of-the-art approaches. We performed a sensitivity study for FLORIA and also demonstrated that FLORIA can be applied to guiding cache partitioning to improve overall system performance. FLORIA is publicly available at https://github.com/yaochengx/FLORIA.

8 ACKNOWLEDGEMENT

We thank all the anonymous reviewers for their constructive feedback. The research is supported in part by the National Key R&D Program of China under Grant No. 2022YFB4500701, and the National Science Foundation of China (Grants No. 62032001 and No. 62032008).

REFERENCES

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, May 1989.
[2] Michael Badamo, Jeff Casarona, Minshu Zhao, and Donald Yeung. Identifying power-efficient multicore cache hierarchies via reuse distance analysis. *ACM*

Transactions on Computer Systems (TOCS), 34(1):3, 2016.
[3] Nathan Beckmann and Daniel Sanchez. Modeling cache performance beyond lru. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236. IEEE, 2016.
[4] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 169–180. ACM, 2005.
[5] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. Optimal cache partition-sharing. In *2015 44th International Conference on Parallel Processing*, pages 749–758. IEEE, 2015.
[6] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for fdo compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52, 2010.
[7] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 308–319. ACM, 2013.
[8] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, 2010.
[9] Perf developers. *perf_event_open - Linux man page*.
[10] Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors*, February, 2015.
[11] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM Sigplan Notices*, volume 38, pages 245–257. ACM, 2003.
[12] David Eklov and Erik Hagersten. Statstack: Efficient modeling of lru caches. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 55–65. IEEE, 2010.
[13] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117. IEEE, 2018.
[14] Andrew Hilton, Neeraj Eswaran, and Amir Roth. Fiesta: A sample-balanced multi-program workload methodology. *Proc. MoBS*, 2009.
[15] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 351–364, 2016.
[16] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design*, pages 629–636. IEEE, 2015.
[17] Yunlian Jiang, Eddy Z Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *International Conference on Compiler Construction*, pages 264–282. Springer, 2010.
[18] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: principled and practical scheduling for serverless functions. In Ada Gavrilovska, Deniz Altinbüken, and Carsten Binnig, editors, *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, pages 289–305. ACM, 2022.
[19] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.
[20] Xu Liu and John Mellor-Crummey. Pinpointing data locality bottlenecks with low overhead. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–193. IEEE, 2013.
[21] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.
[22] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.
[23] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.
[24] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. Establishing a base of trust with performance counters for enterprise workloads. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 541–548, 2015.
[25] Cedric Nugteren, Gert-Jan Van den Braak, Henk Corporaal, and Henri Bal. A detailed gpu cache model based on reuse distance theory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 37–48. IEEE, 2014.
[26] Moinuddin K Qureshi and Yale 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)*, pages 423–432. IEEE, 2006.
[27] Ashay Rane and James Browne. Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 147–156. IEEE, 2012.
[28] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.
[29] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 53–64. ACM, 2010.
[30] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. Phase guided profiling for fast cache modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 175–185. ACM, 2012.
[31] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. Locality approximation using time. In *ACM SIGPLAN Notices*, volume 42, pages 55–61. ACM, 2007.
[32] SPEC. *SPEC CPU Benchmarks*.
[33] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on computers*, 41(9):1054–1068, 1992.
[34] G Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 323–334. ACM, 2014.
[35] G Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.
[36] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. *ACM SIGARCH Computer Architecture News*, 37(1):121–132, 2009.
[37] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient {MRC} construction with {SHARDS}. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 95–110, 2015.
[38] Qingsen Wang, Xu Liu, and Milind Chabbi. Featherlight reuse-distance measurement. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 440–453. IEEE, 2019.
[39] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 335–349, 2014.
[40] H. Wong. *Intel Ivy Bridge cache replacement policy, http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/*.
[41] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. Hotl: a higher order theory of locality. *ACM SIGPLAN Notices*, 48(4):343–356, 2013.
[42] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. Dcaps: dynamic cache allocation with partial sharing. In *Proceedings of the Thirteenth EuroSys Conference*, page 13. ACM, 2018.
[43] Jun Xiao, Andy D. Pimentel, and Xu Liu. Cppf: A prefetch aware llc partitioning approach. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.
[44] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.