Lightweight Detection of Cache Conflicts

Probir Roy
College of William and Mary
USA
proy@cs.wm.edu

Shuaiwen Leon Song
Pacific Northwest National Laboratory,
College of William and Mary
USA
shuaiwen.song@pnnl.gov

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
USA
sriram@pnnl.gov

Xu Liu
College of William and Mary
USA
xl10@cs.wm.edu

Abstract

In memory hierarchies, caches perform an important role in reducing average memory access latency. Minimizing cache misses can yield significant performance gains. As set-associative caches are widely used in modern architectures, capacity and conflict cache misses co-exist. These two types of cache misses require different optimization strategies. While cache misses are commonly studied using cache simulators, state-of-the-art simulators usually incur hundreds to thousands of times a program's execution runtime. Moreover, a simulator has difficulty in simulating complex real hardware. To overcome these limitations, measurement methods are proposed to directly monitor program execution on real hardware via performance monitoring units. However, existing measurement-based tools either focus on capacity cache misses or do not distinguish capacity and conflict cache misses. In this paper, we design and implement CCProf, a lightweight measurement-based profiler that identifies conflict cache misses and associates them with program source code and data structures. CCProf incurs moderate runtime overhead that is at least an order of magnitude lower than simulators. With the evaluation on a number of representative programs, CCProf is able to guide optimizations on cache conflict misses and obtain nontrivial speedups.

CCS Concepts • **General and reference** → **Measurement**; **Metrics**; **Performance**;

Keywords cache conflict, cache miss, profiling, optimization

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168819

ACM Reference Format:
Probir Roy, Shuaiwen Leon Song, Sriram Krishnamoorthy, and Xu Liu. 2018. Lightweight Detection of Cache Conflicts. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3168819

1 Introduction

Modern CPU architectures employ multiple levels of caches to minimize the costly data accesses to main memory. Any reference to main memory brings a chunk of consecutive data to cache. This sequence of bytes, referred as a cache line, is a unit of cache-memory mapping. Mapping memory lines to cache lines can follow one of three strategies: direct mapping, fully-associative mapping, and set-associative mapping. In the direct-mapping strategy, a cache line is mapped to a fixed index in the cache, potentially resulting in poor utilization of the total cache. Although the fully-associative strategy increases the cache utilization by a flexible line placement, it has high implementation overhead and energy consumption. To overcome the limitations of these two approaches, modern CPU architectures usually employ set-associative mapping.

An N-way set-associative cache consists of a number of sets where each set can hold N cache lines. The set-association of a cache line is determined by the index bits of the referenced address. The tag bits of a line address determines the cache line within specific cache set. The offset bits point the byte position within the cache line. Figure 1 shows an example of set-associative mapping of cache lines. In comparison to direct-mapping strategy, set-associative mapping has higher cache utilization as multiple addresses can be mapped to same cache position. Furthermore, set-associative cache outperforms fully-associative cache as an N-way set-associativity can hold up to N addresses at a time with same index bits; resulting in an increased re-usability than the fully-associative cache. Despite such advantages, the set-associative cache often suffers from certain memory access patterns that cause uneven utilization of cache sets resulting in excessive cache misses.

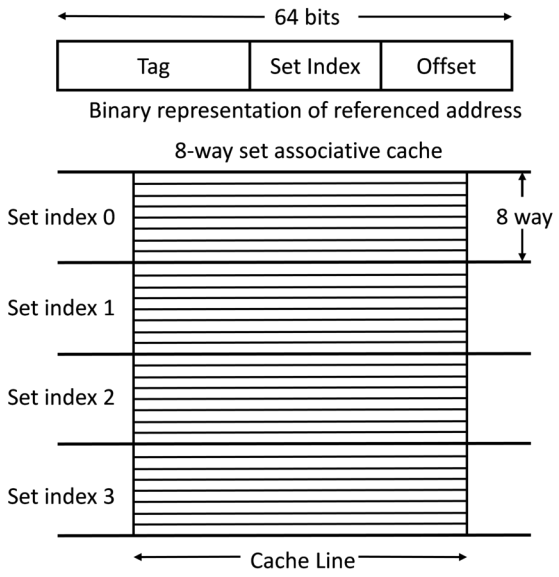

Figure 1. Obtaining cache set index from reference address.

Classical cache models [38] categorize three types of cache misses: cold, capacity, and conflict misses. Cold misses occur when accessing the data for the first time. Capacity misses happen because of the limited cache size. It can be modeled by reuse distance [4], which is defined as the number of unique cache lines accessed between the use and reuse of a single cache line. If the reuse distance is larger than the cache capacity, a capacity miss occurs at the reuse point. Conflict misses occur because of set conflicts in set-associative caches. Because the number of cache lines per set is limited, frequently referencing different data mapped to the same cache set causes cache eviction. Since cache conscious applications largely reuse data from the cache, frequent conflict misses cause immense performance degradation.

Compiler-based techniques utilize the knowledge about the underlying cache architecture and compute the mapping of the array on the set-associative cache. Conflict misses are typically identified by cache simulators. Although simulators can pinpoint conflicts in cache sets, they suffer from both high runtime and space overhead. Moreover, it is difficult to accurately, thoroughly simulate caches in modern CPU architectures. Unlike cache simulators, we present CCProf, an efficient profiler to pinpoint conflict misses from memory access pattern of large-scale applications. For lightweight detection of conflict misses, CCProf depends on the sampling technique supported by performance monitoring units (PMU) in modern CPU architectures. The challenge of using PMU resides in the sparse samples of memory accesses without providing any information directly related to cache conflicts. To address this challenge, we propose a novel metric—*re-conflict distance* (RCD) to identify cache conflicts from sparse PMU samples.

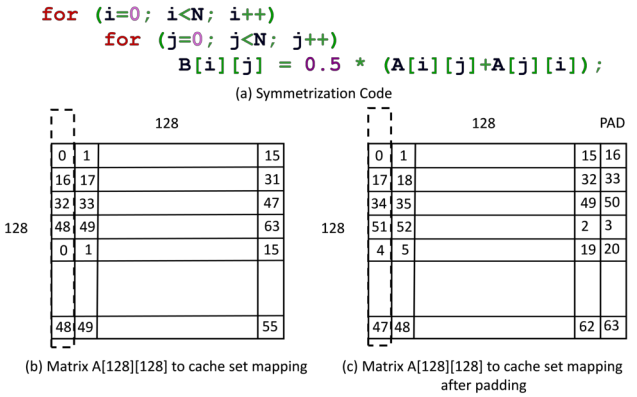

Figure 2. A detailed explanation of cache conflict and optimization of Symmetrization.

As a practical tool, CCProf analyzes fully optimized binary executables without requiring manual instrumentation. Moreover, CCProf is compiler and programming language independent. We evaluate the performance and accuracy of CCProf using 20 applications. Guided by CCProf's analysis, we are able to apply optimization by padding data structures and achieve speedups up to 1.27×. In summary, the main contributions of this paper are:

- We introduce a novel metric—RCD and use it to identify cache conflict misses.
- We adopt a lightweight but lossy sampling mechanism based on hardware PMU and provide a statistical approximation of RCD metrics to pinpoint conflict misses.
- We implement CCProf that reports significant conflict misses and attributes them back to responsible loops and data structures. We verify the correctness of CCProf by comparing with simulation results.
- With the guidance of CCProf, we are able to optimize several programs and obtain nontrivial speedups.

2 Background

2.1 Cache Conflict Misses

We explain the source of conflict misses via an analytical simulation of cache access patterns. Figure 2 illustrates an example code, which performs symmetrization of matrix A in a loop nest. This computation kernel is widely used in quantum chemistry applications [37]. We run the loop nest on a 128×128 matrix. We also run the same code block by adding a pad of 64 bytes at the end of each row of the matrix. Experimental results show that such padding reduces L2 cache misses by up to 91.4%. This performance improvement is a result of cache conflict miss reduction and can be explained using mathematical modeling of the execution [16].

For an 8-way set associative cache with 64 sets and 64B cache line, the first row of the matrix is mapped to the first 16 consecutive cache sets and the second row will be mapped

to the next 16 cache sets, and so on. Due to limited cache sets, the first column of the 5th row will be mapped to set 0 again. For an 8-way set associative cache, the loop nest will start to evict the cache line from cache set 0 from the 9th access until it completes the innermost loop. As a result, instruction accessing matrix A by row will see higher cache misses as the cache line will be evicted by instructions responsible for column accesses. As shown in the figure, column access in the inner loop will frequently utilize four cache sets while other cache sets remain underutilized. When a padding of 64 bytes is applied to the row, the mapping between rows of the matrix and the cache set changes. Figure 2-c shows that the new mapping shifted by one set over the original mapping. With this new mapping, a column access of the matrix in the inner loop will be distributed across all the cache sets and increase cache utilization. This utility of padding decreases conflict cache misses in the computation kernel.

2.2 Sampling-based Cache Behavior Analysis

Our proposed CCProf analyzes memory reference pattern to detect cache conflict misses in the applications. This analysis largely depends on four pieces of information of individual cache references: (1) instruction pointer, (2) data address, (3) type of cache reference (i.e., hit, miss) and (4) level of cache (i.e., L1 cache). Fortunately, modern CPU architectures provide this information through the performance monitoring units (PMU). PMUs of these architectures support event sampling, a mechanism that collects memory reference information at a predefined sampling period. For instance, Precise Event Based Sampling (PEBS) [21] supported by Intel architectures provides all these information that CCProf requires to pinpoint cache conflicts.

PEBS supports address sampling, a type of event-based sampling that allows associating sampled performance events with instruction pointers (IP) and effective data addresses. While instruction pointers can be mapped to the source code, a sequence of sampled data addresses also carry information about data reference patterns of the monitored application. Moreover, PEBS address sampling in recent Intel architectures (i.e., Haswell and its successors) allows precisely monitoring cache misses at memory level. Address sampling allows low-overhead performance analysis. As hardware profiling relies on hardware interrupts to capture snapshots of an instruction execution, it is lightweight compared to other software-based sampling methods. Despite these advantages of hardware-based address sampling, none of the current PMUs identifies cache conflict misses. Additionally, a valid inference of application characteristics from sampled information is challenging due to lossy data collection from sparse samples. To address this challenge, a novel technique is required to infer access/miss patterns. CCProf utilizes a novel metric, *Re-Conflict Distance* (RCD), to estimate memory miss patterns with lossy information

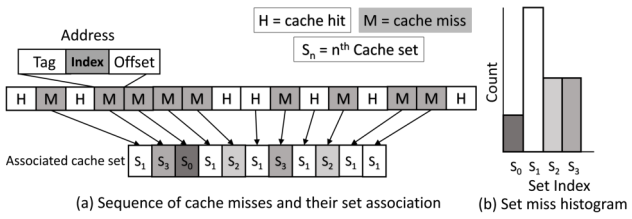

Figure 3. Sequence of cache set miss and their distribution.

from samples. Furthermore, CCProf relies on statistical inference to accurately identify significant conflict misses from memory miss pattern analysis.

3 Methodology

In this section, we describe the profiling technique of cache conflict detection. Identification of conflict misses largely depends on cache miss pattern analysis. In particular, we monitor a sequence of cache misses of the target application. We attribute each sampled cache miss to the corresponding cache set and retrieve a series of sets from the reference pattern, as shown in figure 3-a. We further analyze this sequence to infer probable conflict misses based on the following observation:

> *Observation 1*: A relatively larger portion of cache misses in a subgroup of the total cache sets over the others indicates conflicts in those cache sets.

This observation is intuitive: if no cache conflict misses occur, each cache set should receive similar amount of misses. Thus, we use the term conflict misses to denote imbalanced cache set utilization. We refer to a cache set with high conflicts as a victim set. Figure 3-b illustrates the histogram of the associated cache sets with the miss sequence. Apparently, the histogram illustrates an underlying imbalance of cache set utilization. The sequence in Figure 3-a causes the highest four eviction on set S1, while set S0 is evicted only once. If such imbalanced cache set utilization consistently appears throughout the entire program execution, it can significantly degrade the performance.

To identify the imbalanced cache set utilization in a cache miss sequence and provide optimization guidance, CCProf adopts a series of techniques. Section 3.1 describes the scheme that CCProf uses to associate misses with cache sets; Section 3.2 defines a new metric—RCD that quantifies the imbalance; Section 3.3 discusses the measurement of RCD. Section 3.4 shows the necessary information CCProf collects to guide code optimization on cache conflicts.

3.1 Cache Set Attribution

CCProf profiles the target application to capture cache misses and collects effective data address using address-sampling. CCProf associates each sampled miss with the corresponding

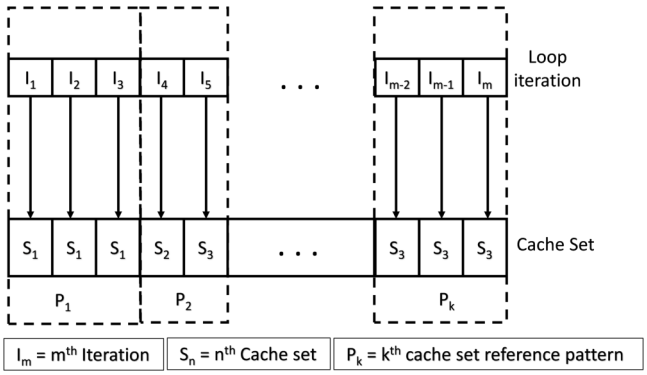

Figure 4. Nonuniform pattern of conflicting cache set access.

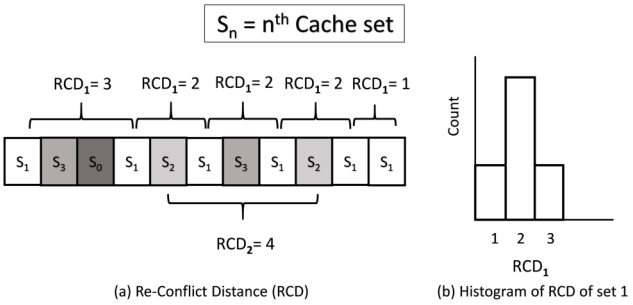

Figure 5. Re-Conflict Distance (RCD) and its distribution.

set in cache using the effective address. As shown in Section 1, the set association of a cache line is determined by the index bits of the effective address. CCProf analyzes L1 cache misses; modern implementations of L1 caches are virtually indexed and physically tagged (VIPT). CCProf retrieves the index bits from the virtual address of reference captured along with the cache miss instruction [1].

3.2 A New Metric for Imbalanced Miss Pattern

Although a set distribution shown in figure 3-b can describe the set utilization of a sequence of cache misses, it does not contain temporal information of the access pattern. Temporal information holds the locality signatures of conflict misses. Depending on the memory access pattern in a loop, a cache set can be the victim of the conflicts for a series of iterations and can be well utilized for others. Figure 4 illustrates such locality signatures in a loop where iterations I1 to I3 conflict at set S1, next iterations I4 to I5 conflicts on set S2 and S3, and so on. To identify this locality of victim sets, we introduce Re-Conflict Distance (RCD), a program and architecture independent metric that captures the temporal pattern of a sequence of cache set misses.

[1]Since L2 and LLC caches are physically indexed, a virtual to physical address mapping is required to retrieve index bits for profiling higher level of cache. But that consideration is out of scope of this paper.

Definition 1 (Re-Conflict Distance (RCD)). *We define Re-Conflict Distance of a cache set (S) for a program context (P) (i.e., loops, functions) as the number of intermediate cache misses between two consecutive cache misses on the set S.*

Figure 5-a shows the Re-Conflict Distance of cache set 1 for the sequence of cache misses. With the help of relative distance of cache miss appearances, RCD quantifies the temporal pattern of the associated sets of cache misses. For example, a change of set S1's RCD from 3 to 1 indicates a more frequent use of the set. This feature of RCD comes with the following useful property which eventually can explain and quantify cache conflicts in a program context.

> *Observation 2*: If an application has no cache conflict misses, the Re-Conflict Distance (RCD_s) of each set (S) will be equal to the number of cache sets (N) available on the cache. If $RCD_s < N$, then set S is a victim of imbalanced cache utilization and may incur frequent conflicts.

Intuitively a program context with lower RCD across sets indicates a conflict on the cache sets. A distribution of re-conflict distance captures the miss patterns across cache sets. Figure 5-b illustrates the histogram of RCD of set 1 for the sequence of conflict misses in figure 5-a. Due to the skew on RCD from 1 to 3, set 1 incurs higher conflict misses compared to the other sets and it is a victim set caused by imbalanced cache utilization.

3.3 Measuring Re-Conflict Distance

Effective RCD measurement requires two pieces of information about a memory access pattern:

- The sequence of cache misses.
- The cache set association of each cache miss.

The first piece of information can be accurately acquired by observing the memory behavior of the application on a cache simulator [11]. The cache simulator collects every memory reference of an application. For the second piece of information, it tracks the cache lines held within a cache set. A new cache reference is matched with the list of residing cache lines in the associated set by tag bits. If no hit is found, the cache reference is considered as a cache miss. Once a cache miss is identified, the cache set of the missed memory reference can be retrieved from the index bits of the data address as discussed in section 3.1. From the sequence of associated sets, an accurate RCD can be acquired. Re-conflict distance does not require the knowledge of the cache miss type for operation. RCD does not distinguish between a capacity miss and a conflict miss; instead, frequent capacity misses on a subset of cache are also considered as conflicts on the set and treated as victim sets.

As simulators require detailed memory access information to determine cache misses, simulation based RCD detection incurs substantial overhead and is challenging for analyzing large-scale applications. For this reason, CCProf uses

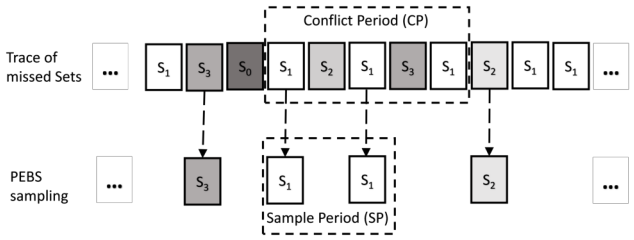

Figure 6. Conflict period and PEBS sampling period.

lightweight address sampling to measure the RCD. Unlike simulation, a measurement-based technique like hardware PMU sampling can directly monitor cache misses. However, despite being lightweight, the lossy nature of address sampling makes it difficult to accurately measure the RCD of cache sets from a sequence of sampled cache misses.

To address this issue, we propose an approximation method to calculate RCD through sampling. Due to the nature of sparse information from PMU samples, we apply inferential statistical analysis to approximately infer the RCDs across cache sets. As the first step, we define the conflict period (CP) of a cache set as the period of consecutive same value of RCD. Figure 6 demonstrates the CP of a cache miss sequence. Applications with a longer CP in a program context indicates a consistent cache behavior for a longer time interval. To effectively identify a victim cache set using sampling, CP is required to be long enough. To be precise, CP needs to be larger than the sampling period (SP). Although adjusting to a higher sampling frequency can increase the probability of detecting victim cache sets, from the discussion in Section 2.2, it is univocal that address sampling cannot avoid the information loss. Moreover, while a high-frequency sampling can reduce error in RCD approximation, it also increases the overhead of the analysis.

Fortunately, RCD derived from address sampling holds the property of original RCD. Once CCProf derives approximate RCD of cache sets, it applies statistical inference to detect significant conflict misses in the program context. This statistical inference relies on the following observation:

> *Observation 3*: Applications suffering from imbalanced cache set misses will have a significant amount of short RCDs.

This observation is intuitive: a large number of conflict misses result in significantly imbalanced cache set misses. With an appropriate sampling frequency setting, it is possible to approximate the existence of prominent cache sets in the program context that holds the property of imbalanced cache miss pattern—short RCD. Detection of conflict misses with a higher sampling frequency yields higher accuracy but suffers from increased overhead. Section 5.3 discusses this pair of trade-off with experimental data.

To define a short RCD we rely on an empirical threshold, T. If there is high contribution of L1 cache misses for the approximated RCD shorter than T, it indicates the existence of significant cache conflicts. We define a metric, contribution factor, as the percentage of cache misses of RCD shorter than T over the total cache misses in a program context. A contribution factor, cf_x^p for cache set (x) is measured from the address samples in a program context (p) with Equation 1. In the equation, $N^p_{RCD_x^T}$ denotes the number of samples with RCD shorter than T in set x in the target program context p and N^p_{total} denotes the total sampled cache misses in program context p.

Table 1. Inferring cache utilization from RCD and their contribution to L1 cache misses.

RCD	L1 cache miss contribution	Implication
low	low	Insignificant impact on program context
low	high	Strong indication of imbalanced cache utilization
high	low/high	No indication of unbalanced cache utilization

$$cf_x^p = \frac{N^p_{RCD_x^T}}{N^p_{total}} \quad (1)$$

3.4 Optimization Guidance on Conflict Misses

CCProf uses both approximated RCD and contribution factor to determine existence and significance of conflict misses in the application as shown in Table 1. Applications with high cf under low RCD are worthy of further investigation.

To provide intuitive optimization guidance, we formulate the conflict miss identification as a classification problem: with given L1 cache miss contribution under an RCD threshold, does a loop suffer from conflict misses? To answer this, we use a supervised learning technique where we train a regression model to classify the conflict misses. The model that describes this relation is called simple logistic regression [35]. By definition, simple logistic regression models a relation that has one independent variable (i.e., contribution factor) and one dependent variable with a binary outcome (i.e., conflict miss or no conflict miss). We train this simple logistic regression model to predict possible classification based on a given L1 cache miss contribution under an RCD threshold. It is worth noting that the accuracy of the model highly depends on the sampling period. In Section 5.2, we evaluate the accuracy of our classification model.

Once CCProf determines the existence of conflict misses in a program, it provides more information to guide source code optimization. CCProf performs both code- and data-centric attribution to associate conflict misses with source code. CCProf's code-centric attribution associates every sample with code line and loop. Loops involving a large number of samples are considered hot loops and worthy for further analysis. The code-centric attribution allows programmers to identify code regions with significant conflict misses and

avoid unnecessary optimization efforts on trivial code regions.

CCProf's data-centric attribution identifies responsible data structures for cache conflicts. To achieve this capability, CCProf stores every memory allocation along with their start and end addresses. Once conflicts are identified, samples involved in the conflicts are mapped to the associated memory region using the effective address captured in each sample. Data structures with a large number of conflicting samples are responsible for cache conflicts. The data-centric attribution helps programmers focus the optimization on problematic data structures only.

4 CCProf Implementation

CCProf's conflict detection is a two-step process: online profiling and offline analysis. CCProf's online profiler runs along with binary executables and collects runtime samples with low overhead. CCProf's offline analyzer post-processes the collected data, evaluates dominant cache set conflicts, and attributes them to source code and data structures. The remaining of this section elaborates on the online profiling and offline analysis implementation of CCProf.

Online Profiling CCProf uses libmonitor [24] to preload online profiler within the same address space of the executable binary. As libmonitor captures process and thread creation, CCProf sets up the profiling configuration for each thread/process and monitors them individually, including setting up PMU sampling with event MEM_LOAD_UOPS_RETIRED:L1_MISS and installing sample handler. Upon each sample event, CCProf's sample handler randomly sets the next sampling period based on given probability distribution. Moreover, libmonitor captures the memory allocation functions for CCProf to record data ranges allocated during program execution; this information is useful for data-centric attribution. Finally, CCProf's online profiler serializes the profiles from different threads and writes them into a log file for offline analysis.

Offline Analysis CCProf retrieves the control flow graph (CFG) of the target application from the machine code and uses interval analysis [14] to identify loops. The offline analyzer parses the collected samples to compute the approximate RCD. Finally, offline analyzer's classifier provides binary output stating the existence of cache conflict misses in the target loop.

5 Evaluation

We evaluate CCProf on Intel Broadwell machine with 2.00GHz Xeon E7-4830v4 processors and Intel Skylake machine with 3.50GHz E3-1240v5 processors. Both of the machines are equipped with 32KB private L1 cache and 256KB private L2 cache for each core; Intel Broadwell has 35MB of shared LLC cache while Intel Skylake has 8MB of shared

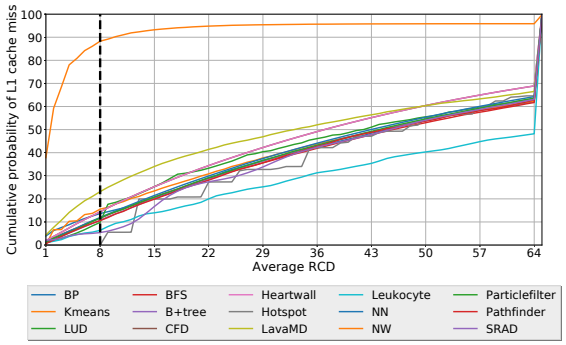

Figure 7. L1 cache miss contribution of RCD in Rodinia benchmark.

LLC cache. Intel Broadwell has 14 cores per socket and Intel Skylake has 4 cores per socket. Both machines have two SMT threads per Core. To evaluate speedups and cache performance, we run parallel applications with 28 and 8 threads on Intel Broadwell and Skylake, respectively, occupying all available threads. Throughout the evaluation section, we measure the RCDs on the L1 cache, which is 8-way set-associative with total 64 cache sets. To validate the measured approximate RCDs, we compare the results with a cache simulator. For this purpose, we collect memory access trace of the target program using Pin [33]. To find cache misses from memory trace we simulate the underlying cache using trace-driven Dinero IV [11] uniprocessor cache simulator.

To evaluate the correctness and overhead of CCProf's RCD-based conflict miss classification, we study applications from Rodinia [8] and PolyBench/C benchmark suite [40]. We compile each application with GCC 4.8.5 and 5.4.0 on Intel Broadwell and Skylake, respectively. Moreover, for accurate code-centric attribution of samples, we disable compiler optimization with (-O0), although we use highest level optimization (-O3) while reporting speedups. Due to the low overhead of the address sampling mechanism, each application is run to completion when evaluating using CCProf. On the other hand, as simulations involve high overhead, we only selectively trace and simulate hot loops identified by address sampling.

5.1 Contribution of RCDs

Section 3.4 introduced contribution factor, cf_x^p as an indicator for conflict miss. Figure 7 plots the cumulative distribution function (CDF) of RCD samples of 18 applications from Rodinia Benchmark [8]. Figure's x-axis presents the sampled RCDs in an increasing order. In each significant loop, we measure the average contribution factor, cf of the RCDs across all cache sets and compute the cumulative probability of L1 cache misses with the increasing order of RCDs. Loops with a high number of conflict misses show high cumulative probability at small RCD. The CDF plot indicates that samples of the Needleman-Wunsch application are not proportionally distributed; a significant amount of L1 cache

misses at short RCD are observed, i.e., RCD of shorter than eight accounts for 88% of the L1 cache misses, while the remaining 12% of the L1 cache misses have RCD longer than eight. CCProf observes frequent cache set misses on the same sets in Needleman-Wunsch, indicating high conflict misses. Cache simulator also confirms that the target loop in Needleman-Wunsch suffers from conflict cache misses. On the other hand, CCProf reports that the hot loops from the rest of the Rodinia benchmarks have little L1 cache miss contribution by shorter RCDs; i.e., only 10-20% of the cache misses occurring when RCD less than eight, while remaining 80-90% of the cache misses are due to the RCD larger than eight. This result indicates a relatively uniform distribution of L1 cache miss in all the cache sets. Cache simulation confirms that these benchmarks do not suffer from conflict misses.

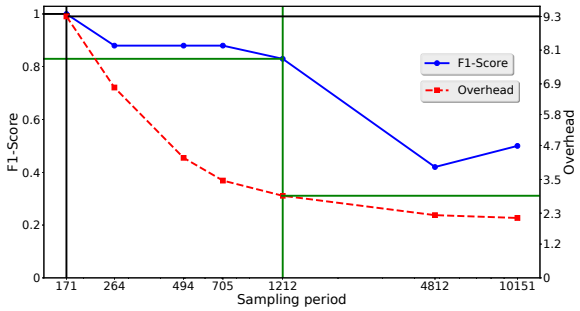

Figure 8. F1-score and average runtime overhead over a range of sampling period.

5.2 CCProf's Accuracy

In section 3.4, we have discussed that for an accurate classification of conflict misses from approximate RCD, we leverage a supervised learning technique. For this purpose, we train a simple logistic regression model that infers the existence of cache conflict in the program context based on the L1 cache miss contribution factor, cf_x^p of RCDs under a predefined RCD threshold. As CCProf approximates the RCD measurement by bursty sampling, the accuracy of the trained model highly depends on the sampling frequency. In this section, we validate the accuracy of the approximate RCD-based regression model with a *simulator*. We assume the cache simulator accurately simulates the L1 cache of a single core in Intel Broadwell and provides the accurate RCDs. The regression model trained with accurate RCDs provides the ground truth. We further modify the simulator to simulate CCProf's event-based sampling by applying random sampling on the traced cache misses at various sampling frequency distributions. At each frequency distribution, the synthesized CCProf simulator provides approximated RCDs of the program context. Due to CCProf's precise event-based sampling, a regression model trained on the synthesized RCDs accurately simulates the CCProf's regression model at a given sampling frequency distribution. Finally, we compare the classification accuracy of the synthesized trained models with the ground truth model.

We train the simple logistic regression model with 16 representative loops where eight of them suffer from cache conflicts, while the rest do not. We trace the loops using Pin and identify the cache misses and their cache set association on Dinero IV cache simulator. We use the contribution factor below RCD of eight as the determinant for the classification. We evaluate the classification model using k-fold (e.g., 8-fold) cross-validation method. We measure the classification accuracy of the trained models using F1-score. In statistical analysis, F1-score is a measure of test accuracy [42]. It is the harmonic mean of precision and recall. With a high F1-score, a model has high precision and recall, implying higher accuracy. The best classifier (e.g., ground truth) results in an F1-score of 1 whereas worst result is 0. Figure 8 plots the F1-score of the classifier at different sampling period distributions. From the figure, smaller sampling period results in better model accuracy; with the given 16 applications, the simple logistic regression model, trained on approximate RCD, can achieve the best F1-score—1 at a mean sampling period of 171.

5.3 CCProf's Overhead

Although small sampling period achieves high accuracy, this also causes high runtime overhead. We further measure the CCProf's sampling overhead at different frequencies. Figure 8 plots the average runtime overhead of CCProf along with accuracy at a various sampling period. CCProf can achieve an F1-score of 0.83 at the mean sampling period of 1212 where the runtime overhead is 2.9×. For the target applications, CCProf can gain the best F1-score with 9.3× average runtime overhead. Comparatively, the simulation based method has an average overhead of ~1000× to trace the target loops; higher for whole application tracing. Considering the heavy-weight memory tracing of simulator-based methods, CCProf has negligible overhead. Table 2 compares CCProf's sampling overhead with simulation overhead of six case studies. Simulation-based conflict detection incurs a median overhead of 264× for the target loops. From the table, number of active loops are much higher and will indeed cause larger overhead for the whole application simulation. Comparatively, CCProf has a small median overhead of 1.37× for whole application profiling, proving usability of the approach. Based on this study, we recommend using a mean sampling period of 1212. For more accurate results, users can configure CCProf at a higher sampling frequency inducing larger overhead.

Table 2. Benchmarks and CCProf performance.

Application	# of active inner loops	Target loop contribution	Simulation overhead (Loop)	CCProf overhead (overall)
NW [8]	11	29.51%	60.4×	1.25×
MKL FFT [19]	12	50%	15.83×	1.41×
ADI [40]	5	80%	326.33×	1.51×
Tiny_DNN [1]	41	19.4%	202.33×	1.1×
Kripke [25]	19	5.1%	650.74×	1.32×
HimenoBMT [15]	2	99%	4664.67×	27×

Table 3. Speedup and cache miss reduction after optimization.

Application	Intel Broadwell - 28 Threads (14 core×2 SMT/core)				Intel Skylake - 8 Threads (4 core×2 SMT/core)			
	Speedup	L1 reduction	L2 reduction	LLC reduction	Speedup	L1 reduction	L2 reduction	LLC reduction
NW [8]	3.03×	3.8%	5.1%	52.7%	1.55×	2.2%	8.5%	20.9%
MKL FFT [19]	1.13×	-0.1%	-13.4%	22.6%	1.03×	1.1%	5.6%	11.7%
ADI [40] (sequential)	1.26×	12.15%	-1.41%	0.21%	1.70×	-27.9%	-45.8%	69.2%
Tiny_DNN [1] (Loop Only)	1.09×	4.2%	65.3%	34.5%	1.24×	-7.8%	27.5%	12.2%
Kripke [25](Loop Only)	94.6×	97.2%	98.3%	92.7%	11.1×	91.7%	93.8%	8.8%
HimenoBMT [15]	1.12×	22.5%	52.4%	5.5%	1.14×	91.2%	42.2%	-6.4%

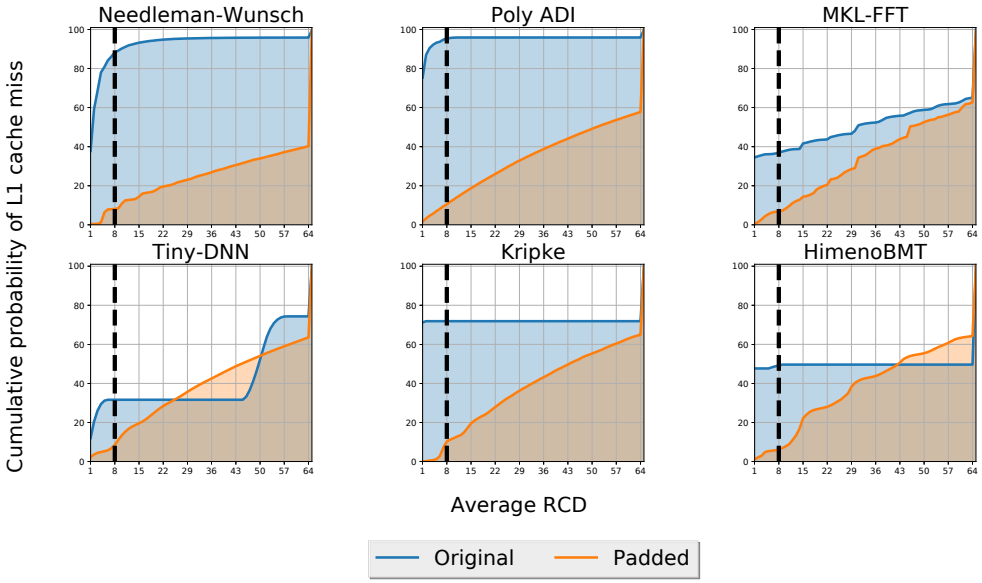

Figure 9. Cache-Conflict measurement by CCProf after optimization.

6 Case Studies

In this section, we evaluate CCProf's utility on a few applications suffering from conflict cache misses. Table 2 summarizes each case with target loop's contribution to L1 cache misses. We further apply optimization by padding on the responsible data-structures pointed by CCProf and report the speedup in Table 3. From the table, conflict miss optimizations achieve nontrivial speedups for the six applications on Intel Broadwell and Skylake. PMU also confirms that the optimization reduces overall cache misses in different levels of caches.

Finally, we run CCProf to classify the optimized code. CDF plot on Figure 9 shows that all the original implementations have a high L1 cache miss contribution under short RCD, but after optimization by padding, shorter RCDs account for a small number of the L1 cache misses; indicating optimization by padding lowered conflict miss. This evidence proves the correctness of CCProf's classification.

In the next section, we look into each application in more details and explain the utility of CCProf.

6.1 Rodinia Needleman-Wunsch

Needleman-Wunsch, prevalent in bioinformatics, is a cache sensitive application from Rodinia Benchmark [8]. This algorithm is used for global optimization of DNA sequence alignment. It applies dynamic programming to deduce the optimal alignment of a pair of DNA sequences organized in

Table 4. Distribution of cache set usage per loop in Needleman-Wunsch.

Loop with line number	L1 cache miss contribution	# of Cache Sets utilized
needle.cpp:289	19.20%	64
needle.cpp:189	29.51%	64
needle.cpp:128	29.60%	64
needle.cpp:138	**10.20%**	**45**
needle.cpp:199	**10.08%**	**41**
needle.cpp:320	0.04%	30
needle.cpp:147	0.44%	19
needle.cpp:208	0.44%	18
needle.cpp:220	0.23%	10
needle.cpp:159	0.25%	9
needle.cpp:273	0.01%	2

```
1 for ( int i = 0; i < BLOCK_SIZE; ++i )
2 {
3   #pragma omp simd
4   for ( int j = 0; j < BLOCK_SIZE; ++j)
5   {
6     reference_l[i*BLOCK_SIZE + j] = reference[max_cols*(
          b_index_y*BLOCK_SIZE + i + 1) + b_index_x*
          BLOCK_SIZE +  j + 1];
7   }
8 }
```

Listing 1. Loop of Rodinia Needleman-Wunsch benchmark causing high L1 cache miss count.

2D-matrix. The algorithm runs on the matrix from top-left to bottom-right and computes the score of each cell based on the previously visited neighboring cells. At the implementation level, Needleman-Wunsch applies loop tiling to increase cache reuse. For this, it copies a tile of large reference matrix into a smaller local matrix, applies optimization locally and finally writes back the result to the reference matrix.

CCProf's code-centric attribution identifies hot loops with significant L1 cache misses in Needleman-Wunsch. Table 4 summarizes the loop contributions and number of cache sets utilization throughout the runtime. From the table, loops `needle.cpp:138` and `needle.cpp:199` utilize only a subset of cache sets (e.g., 45 and 41 out of total 64 sets) and contributes 20% of the total L1 cache misses. Due to their poor cache utilization and high contribution to the L1 cache miss, it can be inferred that those loops incur frequent cache eviction. We further analyze the RCD of other two significant loops `needle.cpp:189` and `needle.cpp:128`. CCProf reports that shorter RCD such as eight and below contributes 88% of the L1 cache misses, indicating high re-conflict on the same cache set. Listing 1 shows loop `needle.cpp:189` as an example. This loop copies the reference array into a local memory for tiling. Furthermore, data-centric attribution identifies that these conflict misses are caused while accessing *reference* matrix and *input_itemsets* matrix in tiles. Due to the alignment of the two matrixes and access pattern, Needleman-Wunsch causes inter-array conflict. CCProf can effectively identify high conflict region similar to simulator-based RCD measurement; but from table 2 CCProf incurs an overhead of only 25% of total runtime.

```
1 //column sweep
2 for (j=1; j<_PB_N-1; j++) {
3  p[i][j] = -c / (a*p[i][j-1]+b);
4  q[i][j] = (-d*u[j][i-1]+(SCALAR_VAL(1.0)+SCALAR_VAL(2.0)
       *d)*u[j][i] - f*u[j][i+1]-a*q[i][j-1])/(a*p[i][j
       -1]+b);
5 }
```

Listing 2. Region of code causing significant L1 cache misses in ADI from PolyBench.

To eliminate the inter-array conflict miss, we pad 32 bytes and 288 bytes at the end of each row of *reference* and *input_itemsets* matrixes. This optimization shows 1.25× and 3.03× speedup for the sequential and parallel run on Intel Broadwell. On Intel Skylake, multi-threaded execution achieves 1.55× speedup. We further verify the source of speedup by measuring cache miss reduction on memory layers. Table 3 shows that Needleman-Wunsch enjoys cache miss reduction in all three cache hierarchies; particularly, LLC cache observes the highest cache miss reduction of 52.7% and 21% on Intel Broadwell and Skylake respectively. Finally, CCProf reports in figure 9 that after optimization, shorter RCD decreases its contribution up to 90% to the L1 cache misses, indicating a reduction of conflict misses.

6.2 PolyBench/C - ADI

PolyBench/C's [40] Alternating Direction Implicit (ADI) is a 2D partial differential equation solver that iteratively solves systems of equations in each direction. At the program level, solver iteratively performs a column sweep and row sweep on a 2D matrix. CCProf's code-centric attribution identifies loop in listing 2 is responsible for 80% of the L1 cache misses in ADI. The data-centric attribution points that matrix u is the victim of conflict miss. The loop accesses matrix u by column and causes frequent conflict on the same set. Both CCProf and simulation confirms the frequent conflict with RCD of 1. We apply a padding of 32 bytes at the end of each row of the matrix. This optimization brings 1.26× and 1.7× speedup on a single core of Intel Broadwell and Intel Skylake respectively. CCProf also confirms that after padding, the shorter RCD (i.e., eight or shorter) reduces L1 cache miss contribution by 89%, denoting no dominant feature of conflict is observable after optimization. Finally, performance metrics from PMU confirms that L1 cache misses reduce by 12.2% on Intel Broadwell. On Intel Skylake, the LLC cache miss reduces by 69%. With such performance benefit, CCProf based conflict detection only adds up a low overhead of 26% of the total runtime.

6.3 MKL- FFT

Intel MKL's Fast Fourier Transform (FFT) [19] routines are highly optimized for Intel CPUs. Cache conflict is a well-known issue for multidimensional Fourier transformation with data of 2-power sizes on each dimension. We evaluated a 2D 4096×4096 DFT. As all the computation is performed

```
for (cnn_size_t c = 0; c < in_size_; c++) {
  a[i] += W[c*out_size_ + i] * in[c];
}
```

Listing 3. Forward propagation of fully-connected layer of Tiny-DNN.

in closed source MKL library, CCProf's cannot attribute the samples to the code but can associate samples to anonymous code blocks. CCProf reports that a loop within MKL library consumes 50% of the L1 cache misses. Furthermore, CDF plot of RCD in figure 9 shows that shorter RCD (i.e., eight or shorter) accounts for 37% of the L1 cache misses in the loop indicating high conflict misses in the code region. After applying optimization by padding 8 element at the end of each row, parallel execution of MKL FFT enjoys a speedup of 1.13× and 1.03× on Intel Broadwell and Intel Skylake. PMU's performance metrics reports that the LLC cache misses reduce by 22.6% on Intel Broadwell. CCProf's effective conflict miss detection incurs only 41% of runtime overhead.

6.4 Tiny-DNN

Tiny-DNN [1] is a header only deep learning library designed for real applications including IoT devices and embedded system. We evaluate the CIFAR model training on CIFAR10 dataset. CCProf identifies that the forward propagation of the fully connected layer suffers from conflict miss. Listing 3 shows the responsible loop that accesses the weight matrix, W in fixed stride. CCProf shows that this strided access causes shorter RCDs while accessing the array in the loop. As the Tiny-DNN is a complex application, we isolate the representative loop for further optimization by padding. With weight array padding, the target loop gains a 1.09× and 1.24× speedup while running in parallel on Intel Broadwell and Skylake. Cache performance analysis also reports that optimizing Tiny-DNN reduces L1, L2, and LLC cache misses by 4.2%, 65.3%, and 34.5% respectively on Intel Broadwell. A similar effect is observed on Intel Skylake. The CDF plot of figure 9 indicates that after padding the loop, the shorter RCD (i.e., eight and shorter) observes 73% reduction in L1 cache miss contribution; indicating a decrease of the same cache set misses. CCProf accounts for only 10% overhead of Tiny-DNN runtime.

6.5 Kripke

Kripke [25] from LLNL is a 3D Sn deterministic particle transport code. CCProf locates conflict misses in the particle edit kernel. As shown in listing 4, this conflict is due to accessing the 3-dimensional *psi* vector in column order. Simply transforming to row-order, these loops gain 94.6× and 11.1× speedup when running on Intel Broadwell and Skylake respectively. PMU measurement on table 3 also reports a reduction of cache misses in all three levels of cache. Finally, CDF plot of RCD shown in figure 9 reports that only 10% of the L1 cache misses of the optimized loop has RCD

```
for(int z = 0;z < num_zones;++ z){
 double vol = sdom.volume[z];
 for(int d = 0;d < num_directions;++ d){
   double w = dirs[d].w;
   for(int g = 0;g < num_groups;++ g){
     part += w * (*sdom.psi)(g,d,z) * vol;
   }
 }
}
```

Listing 4. Particle edit kernel of Kripke.

```
#define MR(mt,n,r,c,d)  mt->m[(n) * mt->mrows * mt->mcols
      * mt->mdeps + (r) * mt->mcols* mt->mdeps + (c) * mt
     ->mdeps + (d)]

#pragma omp for nowait
for(i=1 ; i<imax; i++)
 for(j=1 ; j<jmax ; j++)
  for(k=1 ; k<kmax ; k++){
   s0= MR(a,0,i,j,k)*MR(p,0,i+1,j,  k)
    + MR(a,1,i,j,k)*MR(p,0,i,  j+1,k)
    + MR(a,2,i,j,k)*MR(p,0,i,  j,  k+1)
    + MR(b,0,i,j,k)
    *( MR(p,0,i+1,j+1,k) - MR(p,0,i+1,j-1,k)
    - MR(p,0,i-1,j+1,k) + MR(p,0,i-1,j-1,k) )
    + MR(b,1,i,j,k)
    *( MR(p,0,i,j+1,k+1) - MR(p,0,i,j-1,k+1)
    - MR(p,0,i,j+1,k-1) + MR(p,0,i,j-1,k-1) )
    + MR(b,2,i,j,k)
    *( MR(p,0,i+1,j,k+1) - MR(p,0,i-1,j,k+1)
    - MR(p,0,i+1,j,k-1) + MR(p,0,i-1,j,k-1) )
    + MR(c,0,i,j,k) * MR(p,0,i-1,j,  k)
    + MR(c,1,i,j,k) * MR(p,0,i,  j-1,k)
    + MR(c,2,i,j,k) * MR(p,0,i,  j,  k-1)
    + MR(wrk1,0,i,j,k);

    ss= (s0*MR(a,3,i,j,k)-MR(p,0,i,j,k))*MR(bnd,0,i,j,k);
    gosa1+= ss*ss;
    MR(wrk2,0,i,j,k)= MR(p,0,i,j,k) + omega*ss;
  }
```

Listing 5. Region of code of HimenoBMT with high L1 cache misses.

of eight or shorter; whereas in the original loop the shorter RCD accounts for 71.9% the L1 cache misses. This result indicates that CCProf does not observe conflict misses in the optimized code and for this detection, CCProf has only 32% runtime overhead.

6.6 Riken Himeno CFD - HimenoBMT

HimenoBMT [15] is a benchmark for fluid analysis that solves the Poisson equation using 3D Jacobi method. CCProf identifies that the loop nest of Jacobi, shown in listing 5, consumes 99% of the total L1 cache misses. As the conflict misses move to a different set very frequently, conflict period (CP), introduced in section 3.3, is small. As a result, effective detection of conflict miss pattern requires high-frequency sampling. CCProf can identify these small conflict miss pattern with an overhead of 27×. We optimize the conflict misses by padding the 1st and 2nd dimension of the matrix. Executing the optimized code in parallel gains 1.12× and 1.14× speedup on 28-thread Intel Broadwell and 8-thread Intel Skylake. Cache performance measurement shows that

this performance gain is due to 22-91% reduction of the L1 cache misses.

7 Related Work

7.1 Conflict Miss Detection and Optimization

Cache miss classification approaches can be divided into four major categories: trace-driven simulation, analytical modeling, custom hardware based approach, and PMU-based sampling technique.

Trace-driven Cache Simulators [34], [27], [46] take memory access traces as input and simulate underlying cache to capture the behavior of the trace. This simulation of cache model can provide a theoretically accurate classification of cache misses by tracking memory references and reuse in the trace. In addition to classification, trace-based simulators can attribute the conflict misses to source code and data structures. However, trace-based simulators incur high overhead as it requires to trace every memory access. Moreover, sometimes they require heavyweight binary instrumentation [36] which limit their utility. On the other hand, in this paper, we leverage lightweight hardware PMUs; they readily provide a set of sampled cache misses.

As an alternative to trace-based simulation, several lightweight analytical cache modeling techniques were proposed for cache miss classification [2], [12], [13], [7], [48]. Although analytical classification incurs low overhead, their utility is limited due to complex algorithms and geometric degeneracies. Comparatively, our measurement based approach can analyze any complex application and their interactions with underlying cache-systems at low overhead.

To cope with the limitation of simulator and analytical model-based approaches, several customized hardware-based conflict-miss detection techniques have been proposed [3], [10], [45]. One of these techniques, hardware-based miss-classification table (MST) [9] keeps track of the tags of last cache miss and classify any subsequent miss on the set with the same tag as conflict. MST relies on victim buffer that can be used to classify a subset of conflict misses. All of the hardware-based solutions are emulated on processor simulators and are not supported by major CPU architecture implementations. However, CCProf utilizes the facilities of PMU supported by all major CPU architectures.

Utilizing PMU-based sampling technique, DProf [39] is the most related to CCProf. Similar to CCProf, DProf relies on lightweight event-based sampling and statistical reasoning on the collected data. However, as DProf's classification depends on heuristics, it's utility is limited to a subset of conflict misses. DProf assumes that the workload is uniform throughout the runtime, whereas, applications with the dynamic access pattern are common. Unlike DProf, CCProf applies to a wide range of applications. CCProf's RCD-based approach can effectively capture the temporal signature of conflict misses in dynamic workload at low overhead.

Several temporal signature-based memory behavior predictions were proposed [28], [43], [23]. Hu et. al. proposed three timekeeping metrics to predict conflict misses: reload interval, deadtime, and livetime [17]. Authors show that smaller values of these timekeeping metrics are indicative of conflict cache miss. However, all signature based prediction methods require simulation or custom hardware [18] for accurate measurement of the cache line reuse. Similar to these methods, CCProf relies on novel RCD-metric to capture the temporal signature of the workload while incurring low overhead due to address sampling.

7.2 Profile Guided Performance Analysis

Several profile-guided monitoring tools are proposed for measuring memory performance (e.g., PAPI, OProfile). Existing profile guided analysis tools either depends on software based instrumentation [44] or hardware based event profiling (e.g., Intel PTU [22], Intel VTune [20]) for performance monitoring on applications. In addition to general performance visualizations, tools like HPCToolKit [29], SLO [5], Cache Scope [6] and MACPO [41] perform memory access analysis to provide deep insights into a wide range of performance problem. A group of tools like MemProf [26], HPCToolKit-NUMA [30], ArrayTool [31] provide PMU-based analysis techniques to locate a range of data locality issues. Another group of tools like Scalasca [47], ScaAnalyzer [32] uses profile information from hardware PMU's to identify scaling issues. CCProf provides novel PMU-based analysis methods to identify conflict cache misses.

8 Conclusions

This paper proposes CCProf, a lightweight tool to pinpoint significant conflict misses in applications. For this purpose, we have introduced a novel metric RCD to classify conflict cache misses. Although ideally, RCD based conflict miss classification requires precise cache reference patterns, we have demonstrated that CCProf can leverage the approximate hardware-based profiler to estimate the RCDs. Evaluation results show that CCProf achieves high accuracy in low overhead compared to heavy-weight cache simulation. By the guidance of CCProf, we identify the loops and data-structures responsible for conflict cache misses and optimize them to gain speedups.

A Artifact Description

A.1 Abstract

This artifact contains the source code of CCProf along with six benchmarks demonstrating cache conflict behavior. Additionally, we provide the CCProf guided transformed source code of those benchmarks. This artifact comes with scripts to build CCProf and detect cache conflict on any application. Finally, we provide scripts to reproduce the performance results presented in the CGO2018 paper: *Lightweight Detection*

of Cache Conflicts. Successful building and running of the scripts will reproduce figure 9 and performance statistics(i.e., speed up, overhead, cache miss reduction) of table 2.

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Algorithm: Cache conflict detection**
- **Program: Needleman-Wunsch-Rodinia, PolyBench/C - ADI, MKL- FFT, Tiny-DNN, Kripke and HimenoBMT(All sources)**
- **Compilation: GCC version 4.8.5 or later**
- **Transformations: Transformed code is provided**
- **Run-time environment: Linux kernel version 4.8.0 or later. May require sudo access to install python modules**
- **Hardware: Intel Xeon Skylake(E3-1240v5) or Intel Xeon Broadwell(E7-4830v4) or Intel Xeon Haswell(E5-2699v3)**
- **Run-time state: Artifact is sensitive to cache contentions**
- **Execution: CCProf will profile using Intel-PEBS. Might do process pinning.**
- **Output: All results are generated in *CCPROF_result* directory. CCProf's cache conflict prediction is stored in **result* files. Performance metrics (i.e., speed up, overhead and cache miss reduction) are stored in file *CCProfPerformanceMetrics_table2.txt*. Cumulative-distribution of RCD of original and optimized applications are plotted in **.pdf* files.**
- **Experiment workflow: clone git; run *build.sh*; load env vairables with *~/.bashrc*; run *reproduce_result.sh*; observe performance results.**
- **Publicly available?: Yes**

A.2.2 How Delivered

The artifact is publicly available on Github at: https://github.com/proywm/CCProf.git

A.2.3 Hardware Dependencies

As the CCProf depends on Intel's PEBS hardware events, we evaluate CCProf on Intel Broadwell(E7-4830v4) and Intel Haswell(E5-2699v3) micro-architecture. We recommend using either of these two microarchitectures.

A.2.4 Software Dependencies

We recommend to use a Linux kernel version 4.8.0 or later for PEBS driver support. We build CCProf with GCC version 4.8.5 or later and recommend to do so reproduce the results. CCProf needs python2.7 and python modules (i.e., pandas (0.19.2), numpy (1.12.0), pickleshare (0.7.4), plotly (2.2.1), sklearn, matplotlib (2.0.0)) to be installed. To reproduce the results of the paper, we recommend installing Intel MKL library, Intel ICC compiler and cmake (version 2.8.10 or later). The rest of the software dependencies will be installed by the build script.

A.3 Installation

After cloning the repository, navigate to the base directory and run *build.sh*. This should install CCProf. After successful installation, the script will update the *~/.bashrc* file with CCPROFDIR environment variable pointing to CCProf's installation directory. It will also set HpcToolKit's local installation directory in PATH environment variable. Reload the environment variables by running *source ~/.bashrc* before proceeding to the next step to run the CCProf.

A.4 Experiment Workflow

To reproduce the results of the paper, run *reproduce_result.sh*. This will run all the experiments on the six benchmarks, perform post-processing, run evaluation and finally generate *CCPROF_result/*.pdf*, *CCPROF_result/CCProfPerformanceMetrics_table2.txt* and *CCPROF_result/*result* files.

A.5 Evaluation and Expected Result

CCProf's conflict miss analysis results are stored in *CCPROF_result/*result* files. Compare the CCProf's conflict analysis on original and transformed code. For instance, to compare CCProf's evaluation report on ADI, compare *CCPROF_result/ADI_PolyBench_result* and *CCPROF_result/ADI_PolyBench_Optimized_result* and observe loop level cache conflict prediction. Performance metrics (i.e. speed up, overhead and cache miss reduction) are summarized in *CCProfPerformanceMetrics_table2.txt*. These performance metrics are presented in table 2 of the paper. The generated *CCPROF_result/*.pdf* files plot the figure 9 of the paper.

A.6 Customization

You may prefer to run individual test case or evaluate a new application using CCProf. This section demonstrates how to customize scripts and run an application with CCProf.

To run a new application, navigate to *reproduce_case_studies_of_cgo2018_paper* directory, create an application directory and copy scripts from an example test case directory:

```
1 $cd reproduce_case_studies_of_cgo2018_paper
2 $mkdir App
3 $cp ../ADI_PolyBench/*.sh .
```

Go through the scripts and replace with appropriate path and parameters for the new application.

- set path of the directory of the target application in BENCHMARK_RELATIVE_LOCATION
- set the path of the binary of the target application in BENCHMARK_BINARY*
- set appropriate parameters of the target application

To set sampling period, write to *SampleRateThreshold* as shown in the script:

```
$echo "1212" > SampleRateThreshold
```

Once configured, run ccProf_run_and_analyze.sh for general CCProf analysis report

```
$sh ccProf_run_and_analyze.sh
```

CCProf will profile the target application and preprocess the generated files for post-mortem analysis and run analysis to identify cache conflict per loop.

A.7 Notes

As instruction pointer (IP) varies over compilations and the ambiguity of line numbers(e.g., MKLFFT), it may require to manually set the appropriate loop files to generate *CCPROF_result/*.pdf* files.

Acknowledgments

This research is partially supported by National Science Foundation (NSF) under Grant No. 1618620 and 1464157. This research is also supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under award number 63823 and under the "CENATE" project (award number 66150).

References

[1] [n. d.]. Header only, dependency-free deep learning framework in C++11. *http://github.com/tiny-dnn/tiny-dnn* ([n. d.]).

[2] Anant Agarwal, J Hennessy, and M Horowitz. 1989. An analytical cache model. *ACM Transactions on Computer Systems (TOCS)* 7, 2 (1989), 184–215.

[3] Brian N Bershad, Dennis Lee, Theodore H Romer, and J Bradley Chen. 1994. Avoiding conflict misses dynamically in large direct-mapped caches. In *ACM SIGPLAN Notices*, Vol. 29. ACM, 158–170.

[4] Kristof Beyls and Erik D'Hollander. 2001. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, Vol. 14. 350–360.

[5] Kristof Beyls and Erik H DHollander. 2006. Discovery of locality-improving refactorings by reuse path analysis. In *International Conference on High Performance Computing and Communications*. Springer, 220–229.

[6] Bryan R Buck and Jeffrey K Hollingsworth. 2004. Data centric cache measurement on the Intel Itanium 2 processor. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 58.

[7] Siddhartha Chatterjee, Erin Parker, Philip J Hanlon, and Alvin R Lebeck. 2001. Exact analysis of the cache behavior of nested loops. *ACM SIGPLAN Notices* 36, 5 (2001), 286–297.

[8] Shuai Che et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the 2009 IEEE Intl. Symp. on Workload Characterization (IISWC)*. Washington, DC, USA, 44–54.

[9] Jamison D Collins and Dean M Tullsen. 1999. Hardware identification of cache conflict misses. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 126–135.

[10] Jamison D Collins and Dean M Tullsen. 2001. Runtime identification of cache conflict misses: The adaptive miss buffer. *ACM Transactions on Computer Systems (TOCS)* 19, 4 (2001), 413–439.

[11] Jan Edler. 1998. Dinero IV trace-driven uniprocessor cache simulator. *http://www.cs.wisc.edu/~markhill/DineroIV/* (1998).

[12] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1997. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th international conference on Supercomputing*. ACM, 317–324.

[13] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. 1999. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 4 (1999), 703–746.

[14] Paul Havlak. 1997. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.* 19, 4 (1997), 557–567. https://doi.org/10.1145/262004.262005

[15] Ryutaro Himeno. [n. d.]. Himeno benchmark. *http://accc.riken.jp/en/supercom/himenobmt/* ([n. d.]).

[16] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J Ramanujam, and Ponnuswamy Sadayappan. 2016. Effective padding of multidimensional arrays to avoid cache conflict misses. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 129–144.

[17] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. 2002. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 209–220.

[18] Zhigang Hu, S Kaxiras, and M Martonosi. 2003. Timekeeping techniques for predicting and optimizing memory behavior. In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*. IEEE, 166–485.

[19] Intel. [n. d.]. Intel Math Kernel Library. *https://software.intel.com/en-us/mkl* ([n. d.]).

[20] Intel Corporation. [n. d.]. Intel VTune Performance Analyzer. http://www.intel.com/software/products/vtune.

[21] Intel Corporation. 2010. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, Number 253669-032.

[22] Intel Corporation. 2012. Intel Performance Tuning Utility 4.0 Update 5. https://software.intel.com/en-us/articles/intel-performance-tuning-utility. Last accessed: Aug. 10, 2014.

[23] Mazen Kharbutli and Yan Solihin. 2005. Counter-based cache replacement algorithms. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 61–68.

[24] Mark W Krentel. 2013. Libmonitor: A tool for first-party monitoring. *Parallel Comput.* 39, 3 (2013), 114–119.

[25] AJ Kunen, TS Bailey, and PN Brown. 2015. *KRIPKE-a massively parallel transport mini-app*. Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

[26] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A memory profiler for NUMA multicore systems. In *Proc. of the 2012 USENIX Annual Technical Conf. (USENIX ATC'12)*. Berkeley, CA, USA, 1.

[27] Alvin R Lebeck and David A Wood. 1994. Cache profiling and the SPEC benchmarks: A case study. *Computer* 27, 10 (1994), 15–26.

[28] Wei-Fen Lin and Steven K Reinhardt. [n. d.]. Predicting last-touch references under optimal replacement. *Ann Arbor* 1001 ([n. d.]), 48109–2122.

[29] Xu Liu and John Mellor-Crummey. 2013. Pinpointing data locality bottlenecks with low overhead. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 183–193.

[30] Xu Liu and John Mellor-Crummey. 2014. A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *Proc. of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, USA, 259–272. https://doi.org/10.1145/2555243.2555271

[31] Xu Liu, Kamal Sharma, and John Mellor-Crummey. 2014. ArrayTool: a lightweight profiler to guide array regrouping. In *Proceedings of the 23th International Conference of Parallel Architectures and Compilation Techniques*. Edmonton, Alberta, Canada.

[32] Xu Liu and Bo Wu. 2015. ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 47.

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[34] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. 1992. Memspy: Analyzing memory system bottlenecks in programs. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 20. ACM, 1–12.

[35] John H McDonald. 2009. *Handbook of biological statistics*. Vol. 2. Sparky House Publishing Baltimore, MD.

[36] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[37] NWChem 2017. NWChem. http://www.nwchem-sw.org

[38] David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[39] Aleksey Pesterev, Nickolai Zeldovich, and Robert T Morris. 2010. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*. ACM, 335–348.

[40] L.-N. Pouchet and T. Yuki. [n. d.]. PolyBench/C 4.1. *https://sourceforge.net/projects/polybench/* ([n. d.]).

[41] Ashay Rane and James Browne. 2012. Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In *Proc. of the* 12^{th} *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*. Minneapolis, MN, USA.

[42] C. J. Van Rijsbergen. 1979. *Information Retrieval* (2nd ed.). Butterworth-Heinemann, Newton, MA, USA.

[43] Jude A Rivers and Edward S Davidson. 1996. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Parallel Processing, 1996. Vol. 3. Software., Proceedings of the 1996 International Conference on*, Vol. 1. IEEE, 154–163.

[44] Sameer S. Shende and Allen D. Malony. 2006. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (2006), 287–311. https://doi.org/10.1177/1094342006064482

[45] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. 2008. Adaptive set pinning: managing shared caches in chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 135–144.

[46] Jie Tao and Wolfgang Karl. 2006. Detailed cache simulation for detecting bottleneck, miss reason and optimization potentialities. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*. ACM, 62.

[47] Brian J. N. Wylie, Markus Geimer, and Felix Wolf. 2008. Performance measurement and analysis of large-scale parallel applications on leadership computing systems. *Sci. Program.* 16, 2-3 (2008), 167–181.

[48] Hongli Zhang and Margaret Martonosi. 2001. A mathematical cache miss analysis for pointer data structures. In *SIAM Conference on Parallel Processing for Scientific Computing*. Citeseer.