An Accurate Simulator of Cache-Line Conflicts to Exploit the Underlying Cache Performance

Yukinori Sato(✉) and Toshio Endo

Tokyo Institute of Technology, Tokyo 152-8550, Japan
yukinori@el.gsic.titech.ac.jp, endo@is.titech.ac.jp

Abstract. This paper describes a cache-line conflict profiling method that advances the state of the art performance tuning workflow by accurately highlighting the sources of conflicts. The basic idea behind this is the use of cache simulators as a diagnosis tool for cache-line conflicts. We also propose a mechanism that enables to identify where line conflict misses are incurred and the reasons why the conflicts occur. We evaluate our conflict simulator using some of the benchmark codes used in the HPC field. From the results, we confirm that our simulator can accurately model the cache behaviors that cause line conflicts and reveal the sources of them during the execution. Finally, we demonstrate that optimizations assisted by our mechanism contribute to improving performance for both of serial and parallel executions.

Keywords: Accurate cache simulation · Conflict miss detection · Performance tuning · Array padding

1 Introduction

Recently, compiler technologies have made significant progress in automatic vectorization and thread-level parallel execution techniques. However, further source code refactoring for performance tuning is often required to obtain performance close to the versions manually optimized by expert programmers [15]. Primary sources that cause this inefficiency are derived from memory subsystems composed of caches. To increase effective memory bandwidth, or to reduce the latency for a memory access, we need to make good use of cache memories. However, current compilers are often oblivious to cache-conscious optimizations needed to fully utilize the locality in the application. As shown later in this paper, executable binary generated by a compiler does not always fit well to the underlying cache memories. Mostly this is caused by cache-line conflict misses, which often degrade performance significantly.

In this paper, we strive to eliminate performance degradation or performance variability due to line conflict misses. Modern CPU systems typically have highly associative cache structures to avoid conflict misses as much as possible. One example seen in Intel Sandy Bridge CPU is that the L3 cache is organized as a 20-way associative cache. Even in the lower L1 and L2 caches, their associativity

F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 119–133, 2017.
DOI: 10.1007/978-3-319-64203-1_9

is 8-way. However, in some of applications that intensively access a particular set in the associative cache, the number of elements mapped onto the same set can easily exceed the degree of associativity [5] and cause conflict misses. Since this often impacts on performance seriously, we should avoid it by refactoring the source code.

The actual behavior of cache memories within a real system is normally invisible from software. Hence, we provide a way to diagnose avoidable cache misses and a simple workflow to get rid of them. To diagnose cache behaviors, we propose a cache-line conflict simulator called **C2Sim** and attempt to mimic the occurrence of cache-line conflicts by concurrent dual cache simulations. For accurate line conflict detection, we simulate fully-associative (FA) caches as a subsidiary simulation of the underlying set-associative (SA) caches. Furthermore, to assist a performance tuning workflow, we provide a mechanism that reveals the sources of cache-line conflicts and attempts to ease the actual code modification process.

The primary contributions of this paper are as follows:

- We develop C2Sim for identifying cache-line conflict misses effectively. We show that C2Sim provides practically accurate detection of conflicts based on its advanced cache modeling.
- We present a mechanism that can monitor the actual locations of code where line conflict misses are incurred and the reasons why the conflicts occur.
- We show that cache-line conflict misses can be avoided by padding to the appropriate arrays suggested through our mechanism, and such optimizations also contribute to scalable performance improvements on parallel executions.

2 Modeling Cache Structures in Modern CPUs

Driven by semiconductor technology scaling, capacity and associativity of cache memories is continuously increasing. On the other hand, the complexity of algorithms and applications is increasing year by year, which often makes their memory access patterns complicated. Under these situations, cache memories are desired to be useful in all the situations. However, there are no universal cache structures that can exploit locality of references for everything. This is why the underlying cache performance is sensitive to memory access patterns derived from application-specific characteristics and the underlying hierarchical caches and memories.

Therefore, we need to perform source code refactoring to improve cache performance. However, it is hard for skilled programmers to estimate application-specific cache behaviors and apply these to the performance tuning. One of the solutions for this situation is to build a simulator of modern x86_64 CPU caches for performance tuning. Formulating the cache model as a simulator, we can monitor time-varying behaviors of cache memories, which are normally invisible from software. By analyzing sequences of particular events during the simulation, we can detect how the cache miss occurs and whether it could be avoidable or not.

In this paper, we focus on simulating occurrences of cache-line conflicts and a reasoning mechanism upon it for assisting cache performance tuning. Situations where massive amount of requests to a particular set of a cache causes conflict misses are also called cache thrashing. Since it results in serious performance degradation, it should be avoided as much as possible. While cache thrashing is considered to be obvious only in caches that have low associativity, we reveal that it occurs even in current high associative cache structures. Especially, it is seen in typical scientific computing applications that calculate large multi-dimensional arrays.

Here, we investigate how such conflicts normally invisible from application programmers are detected precisely from actual execution. Collins and Tullsen attempted to identify line conflict misses by storing the history of replaced cache lines using an FIFO for every set of caches [4]. In their method, when the miss to a cache set matches a previously evicted tag stored in the FIFO, then it is identified as a line conflict miss. However, as we show later in this paper, this sometimes leads false detections due to the sensitivity to the number of entries of the FIFO. Since this approach fundamentally includes under- or overestimation of conflicts, this is impossible to detect conflicts precisely. To resolve this issue, we propose a detection scheme based on comparison to FA (fully-associative) cache behavior.

3 Cache-Line Conflict Simulator

3.1 FA Cache Based Conflict Detection

We propose a cache-line conflict simulator called **C2Sim**. The key idea for accurate line conflict detection is to conduct FA cache simulation as side simulation to the baseline cache simulation for the target configuration. We also provide a mechanism that identifies where and why line conflict misses occurs.

A cache-line conflict occurs when the number of accessed data elements mapping to the same set exceeds the degree of associativity. In this situation, the original cache line to be accessed was replaced before it is requested again. Such a cache line conflict miss consequently appears only in SA (set-associative) or direct-mapped structures where the number of cache lines within a set is limited. Here, we define a conflict miss as a miss that could be avoided in the FA cache with the same capacity. Since for FA cache there are no limits of associativity, its behavior is a theoretical upper bound for optimization that completely avoids cache-line conflicts. From this definition, when an access to the FA cache hits whereas that for the SA cache misses, it is classified as a conflict miss. When an access both for the FA cache and the SA cache simultaneously misses, it is classified as a capacity miss. Based on these, we implement an FA cache simulator and compare its behavior with the SA cache simulator having the same capacity so that we can detect conflict misses.

It is believed that simulating actual FA caches that maintains true LRU order incurs much overhead compared with a typical SA cache simulation because FA

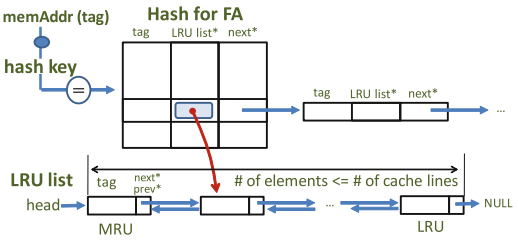

Fig. 1. An efficient FA cache simulation method.

cache needs tag comparison across all entries[1]. To reduce this overhead, we develop an efficient algorithm for FA cache simulation as illustrated in Fig. 1. Instead of using an O(N) or a non-linear algorithm for keeping a true LRU order, we use a hash and a list structure for it. Here, we can directly map a tag to the corresponding entry of the hash with an O(1) lookup unless the entry is heavily shared by other tags (hash collision). If the tag cannot be found in the first entry of the hash, we search the following list structure. After the corresponding tag is found, we have a pointer to the LRU list. This list is implemented using a doubly linked list and maintains a true LRU order. The maximum length of this list corresponds to the number of cache lines and is decided by the capacity of the cache.

3.2 Reasoning Around Line Conflicts

A straightforward conflict detection mechanism discussed above is still insufficient to assist a performance tuning workflow done by programmers. Because the said mechanism only returns how many conflicts appear in the execution, programmers need to read source code carefully and find out where and why the conflicts occur. To improve the productivity of this process, we present a new interface that maps simulation results as clues for performance tuning. We also provide a new mechanism that reveals the locations where and why line-conflict misses occur.

In order to keep track of these, we propose LT-WET (Last Time Who EvicT): a data structure that records key events at instruction level granularity. Figure 2 illustrates how we monitor cache-line conflicts using the LT-WET structure. The key idea here is to store which memory reference instruction triggers a cache-line eviction, and to resolve the reason when a conflict miss for the same set is detected in future. The details are as follows: First, when a cache miss occurs in the SA cache, we store the tag of evicted line to the EvictedTag field and cache miss instruction's address to the Originator field respectively as shown in Fig. 2(a). At the same time, when the memory access is identified as a conflict miss, we search the tag corresponding to the current memory reference instruction from the EvictedTag field. We then identify the miss originator that had

[1] Here, we focus only on true LRU replacement policy for both of FA and SA caches.

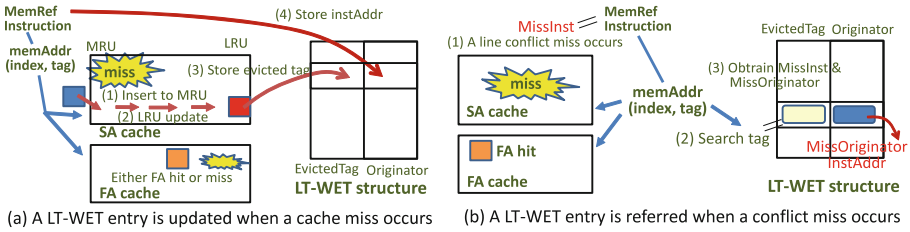

Fig. 2. How to find out where and why the conflict occurs.

caused the last eviction to the current miss as shown in Fig. 2(b). Here, the LT-WET structure is implemented using a hash structure similar to the one used in FA cache simulation to reduce the time for searching tags.

The conflict miss instruction coupled with its originator indicates where and why the conflict miss occurs. The instructions that cause conflict miss can easily be traced back to the source code through debug information that compilers embed inside application binaries.

We also perform memory object relative profiling, which correlates every memory reference to the objects in memory layout appearing in the actual execution. To obtain accessed memory regions, we monitor the ranges (min, max) of accessed memory addresses for all memory reference instructions. Then, each memory region is mapped to a symbol found inside the program. We extract static symbols such as global variables and constants by analyzing the executable-and-linkable (ELF) code. To obtain the symbols of memory regions allocated at runtime, we monitor the memory map at '/proc/pid/maps' for stack regions and hook functions for memory allocators such as malloc. Finally, these memory access related information are consolidated and outputted as results of memory object relative profiling.

3.3 Advanced Cache Modeling for Accurate Simulation

To prevent false conflict detections due to cache modeling inaccuracy, we develop the following advanced simulation mechanisms to C2Sim: a virtual to physical address translation, and a slice mapping mechanism for an L3 cache.

The first mechanism, virtual to physical address translation, enables C2Sim to model Physically Indexed Physically Tagged (PIPT) caches for L2 and L3, and a Virtually Indexed Physically Tagged (VIPT) cache for L1. Here, when the total capacity per associativities is greater than the page size, the mapping for physical address affects cache indexing. This is seen in the typical L2 and L3 caches when using a default 4 KB page. To reflect the actual physical addresses in our simulation, we monitor the mapping table located at '/proc/pid/pagemap' provided by Linux OS when a new page is accessed and record them in a hash table.

The second one, a slice mapping algorithm for L3 cache, is needed to model L3 caches accurately. An L3 cache in modern Intel CPUs is known to be divided

into pieces, usually referred to as slices [6]. The number of slices matches the number of physical cores, and each slice contains 2048 sets, which are equal to 2.5 MB in 20 way set associative configurations. Dividing an L3 cache into slices will spread the traffic almost evenly across the slices and prevent conflicts inside an L3 cache. Therefore, knowing the details of the slice selection algorithm is crucial for building accurate cache simulators. In [6], the authors recover the slice selection algorithm used in modern Intel CPUs based on the Prime+Probe side channel technique. In this paper, following the hash function in [6] (for 8 core CPUs in Table 2), we model the slice selection and mapping mechanism.

As far as we know, C2Sim is the first cache simulator that implements physical address translation and L3 slice mapping algorithm. In Sect. 4, we will validate the accuracy of C2Sim by comparing its cache miss ratio with the one obtained using hardware performance counters in the actual CPUs.

4 Evaluation

4.1 Methodology

In this section, we evaluate our cache-line conflict simulator, C2Sim, using a typical x86_64 Linux server running CentOS 6.7 with two of Intel Xeon E5-2680 CPUs. We implement C2Sim on the top of Pin tool set [9]. For the baseline cache simulation, we set up the same configuration as the underlying CPU, that is L1 = 32 KB 8way, L2 = 256 KB 8way and L3 = 20 MB 20way. Here, we model three level data caches with true LRU replacement where L3 is managed with inclusion policy and L2 is with non-inclusive policy. In the current implementation, C2Sim does not model a shared L3 cache and coherence protocols among different cores, and it just simulates cache behaviors without any delays for cache coherence and communication among other levels.

We use the following benchmarks in HPC field: PolyBench/C 4.2, 3D-FDTD and Himeno benchmark. The PolyBench is a benchmark suite composed of 30 numerical computation kernels in various application domains such as linear algebra computations, image processing, physics simulation, dynamic programming, statistics [3]. Here, we set the data type to double and use LARGE dataset. 3D-FDTD is a benchmark code that evaluates three-dimensional finite-difference time-domain method which is widely used in high-frequency electromagnetic field analysis for the design of electrical devices [10]. Himeno benchmark is composed of a kernel code used in incompressible fluid analysis [2], and one of well-known memory bottleneck applications. We generate executable binary code of these target applications using GNU gcc 4.4.7 with '-O3 -g' option, and first examine cache behaviors for single thread execution. To examine effectiveness for parallel code, we generate multithreaded code with '-fopenmp' option and evaluate their effects for scalability.

In this evaluation, we apply the concept of sampling based cache simulation technique [12] to C2Sim in order to reduce the overheads of on-line cache simulation. Here, we set 100M instructions for the warm-up phase and 500M instructions for the evaluation phase after skipping 4G clock cycles for the first

forward phase. We also note that we turn off the hardware prefetch implemented in the CPU when we examine the cache statistics using our simulator. While the hardware prefetch affects cache behaviors and in most cases results in performance improvements, it sometimes obscures fundamentals of cache conflicts. This is why we turn off the hardware prefetch for evaluating cache behaviors.

4.2 Verification of Our Simulator

Next, we validate accuracy of C2Sim by comparing the cache miss ratio with that from Performance Monitoring Unit (PMU) implemented in the underlying CPU[2]. We note that these cache miss ratios are observed during whole the execution. Here, we use PolyBench suite for this evaluation and measure cache miss ratios during whole the execution. We exclude the programs whose native execution time is less than 0.8 s (gemver, gesummv, atax, bicg, mvt, durbin, trisolv, deriche, jacobi-1d) since a short measurement interval tends to be unsound for sampling simulation.

Table 1. Evaluating our cache simulator with statistics obtained from PMU.

	Native time [s]	C2Sim overheads	PMU			C2Sim			Maximum abs. error
			L1 miss	L2 miss	L3 miss	L1 miss	L2 miss	L3 miss	
floyd-warshall	122.62	27.45	6.24%	50.00%	100.00%	6.25%	50.06%	99.98%	0.06%
correlation	3.98	209.87	66.97%	52.00%	0.00%	66.45%	51.63%	0.02%	0.52%
3mm	5.20	138.85	38.24%	11.00%	1.00%	37.51%	11.12%	0.26%	0.74%
gemm	2.04	81.20	3.15%	100.00%	1.00%	3.13%	100.00%	0.17%	0.83%
ludcmp	39.08	73.50	31.39%	37.00%	22.00%	32.59%	38.28%	20.53%	1.47%
2mm	4.04	150.58	38.06%	12.00%	1.00%	36.27%	11.50%	0.31%	1.79%
covariance	4.00	116.09	67.01%	53.00%	0.00%	66.42%	51.18%	0.02%	1.82%
trmm	1.13	330.67	60.00%	7.00%	2.00%	61.06%	7.62%	0.08%	1.92%
lu	39.55	72.75	29.29%	37.00%	23.00%	29.63%	38.84%	20.97%	2.03%
cholesky	37.55	55.56	25.44%	13.00%	80.00%	25.07%	13.19%	83.20%	3.20%
gramschmidt	3.84	140.91	57.05%	13.00%	1.00%	57.16%	17.20%	0.05%	4.20%
heat-3d	21.08	20.89	9.99%	100.00%	46.00%	6.37%	99.84%	50.66%	4.66%
jacobi-2d	11.29	30.49	3.69%	48.00%	93.00%	8.35%	50.00%	100.00%	7.00%
nussinov	9.36	80.03	34.47%	12.00%	19.00%	34.21%	12.24%	26.34%	7.34%
symm	2.14	110.14	33.96%	29.00%	3.00%	33.36%	21.46%	1.65%	7.54%
fdtd-2d	11.60	26.51	7.91%	85.00%	66.00%	7.73%	87.23%	75.13%	9.13%
syrk	1.16	274.93	23.96%	23.00%	2.00%	23.71%	13.19%	0.26%	9.81%
syr2k	2.29	225.06	33.77%	23.00%	4.00%	33.48%	12.48%	1.61%	10.52%
doitgen	0.92	87.75	33.95%	3.00%	14.00%	34.23%	2.71%	2.80%	11.20%
adi	21.23	37.10	16.77%	43.00%	81.00%	17.03%	30.56%	79.75%	12.44%
seidel-2d	34.98	10.57	8.35%	48.00%	93.00%	3.75%	33.37%	100.00%	14.63%

Table 1 shows the time for native execution of the program and the simulation overheads (slowdown factor) calculated by measuring the time needed for cache

[2] Here, we measure L2/L3 cache miss ratio by Intel PCM, and L1 miss ratio by LIKWID using counters such as L1D_REPLACEMENT and MEM_UOPS_RETIRED.

simulation for the same program. While the simulation overheads have a wide range of variation, the average overhead is 109. To compare the other simulators, we calculate MIPS (Million Instructions Per Second) rates for simulating these programs. The resultant average MIPS rate is 40.06. This is several orders of magnitude faster than typical microarchitectural simulators and the state of the art cache simulators [17].

The result in Table 1 also shows the accuracy of cache modeling on our C2Sim compared with the statistics obtained from PMU. Here, we calculate the average maximum absolute errors of all programs and use it as a criterion for accuracy. The maximum absolute error of each program is obtained by picking up the maximum one among three cache levels after calculating absolute errors between PMU and C2Sim for each cache level. From the results, we find the maximum absolute errors of half of these programs are less than 5%. For all of the programs, the observed maximum absolute error is less than 15%. In average, it is 5.37%. These results indicate that C2Sim is accurate and practical enough to model the performance of hierarchical caches implemented in modern CPUs.

Table 2. Effects of physical address translation and L3 slice mapping.

	VA	PA	VA + slice	PA + slice	PA + slice sampling
Avg. max abs error	48.76%	40.73%	9.33%	5.18%	5.37%

Table 2 shows the accuracy of simulation compared with that of the real machine. Here, VA represents simulations only using virtual addresses; PA represents ones with physical address translation. 'PA + slice (sampling)' indicates typical C2Sim configuration. All of the maximum absolute errors are average of 21 PolyBench programs listed above. From these results, we observe that the L3 slice mapping is an important factor for accurate simulation. It contributes to reducing the average maximum absolute error to 5.37%. It is also observed that even if we enable sampling simulation, the error just increases slightly (0.19%). We also observe that the simulation speed becomes 1.6 times faster if we enable sampling. Therefore, we can understand that coupling these three techniques (PA + slice and sampling) contributes to building an accurate and light-weight cache simulator. These results indicate that our simulator is accurately model the performance of hierarchical caches implemented in modern CPUs.

4.3 Accuracy for Line Conflict Detection

Next, we show the advantages of C2Sim over the existing conflict detection mechanism. Here, we compare the FIFO-based method in [4,17] with C2Sim, where the number of FIFO entries is set to the twice the number of associativities as seen in these papers. Table 3 shows the ratios of line conflicts to the total misses detected in each level. We calculate absolute errors among them and

Table 3. Detected line conflicts in the FIFO-based mechanism and C2Sim.

	FIFO-based			C2Sim			Maximum abs. error
	L1conflict	L2conflict	L3conflict	L1conflict	L2conflict	L3conflict	
gemm	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
covariance	0.00%	88.04%	0.00%	0.00%	88.04%	0.00%	0.00%
correlation	0.00%	88.03%	0.00%	0.00%	88.01%	0.00%	0.02%
doitgen	87.62%	98.90%	0.00%	87.68%	98.90%	0.00%	0.06%
symm	10.12%	37.14%	0.73%	10.04%	37.30%	0.73%	0.16%
2mm	0.29%	0.41%	0.00%	0.00%	0.42%	0.00%	0.29%
gramschmidt	0.00%	84.19%	0.00%	0.00%	84.57%	0.00%	0.38%
syr2k	9.84%	0.26%	2.02%	0.53%	0.27%	2.02%	9.31%
nussinov	21.23%	0.05%	24.78%	0.45%	0.02%	5.89%	20.78%
lu	2.51%	71.45%	21.42%	1.76%	71.58%	0.07%	21.35%
ludcmp	2.51%	71.04%	21.38%	1.74%	71.12%	0.02%	21.36%
3mm	23.17%	0.00%	0.00%	0.00%	0.00%	0.00%	23.17%
trmm	25.75%	2.25%	0.00%	0.62%	2.01%	0.00%	25.13%
syrk	45.08%	0.00%	0.00%	0.06%	0.00%	0.00%	45.02%
floyd-warshall	49.98%	0.00%	0.11%	0.00%	0.08%	0.03%	49.98%
heat-3d	0.00%	47.02%	50.69%	0.00%	0.00%	0.00%	50.69%
fdtd-2d	12.51%	0.00%	73.26%	0.00%	0.00%	5.36%	67.90%
adi	11.91%	6.20%	81.34%	1.09%	6.21%	0.99%	80.35%
cholesky	0.01%	0.12%	87.47%	0.00%	0.51%	0.24%	87.23%
seidel-2d	66.63%	0.00%	95.90%	0.00%	0.00%	0.00%	95.90%
jacobi-2d	50.00%	0.00%	100.00%	0.00%	0.00%	0.00%	100.00%

represent the maximum one across all three level as Maximum abs. error. All the elements are sorted by the field of Maximum abs. error.

From the results, we observe that the FIFO-based mechanism approximates C2Sim's FA-based behaviors in 7 programs (gemm, covariance, correlation, symm, doitgen, 2mm, gramschmidt) with less than 1% absolute errors. However, the rest of them contains a lot of false judgments, where the FIFO-based classifies a miss to a conflict but actually it should be classified to a capacity miss (not to conflict). Since the cache miss behavior of FA caches is a theoretical lower bound that excludes any possible conflict misses, C2Sim can detect the accurate number of cache-line conflicts by combining it with the underlying SA caches.

On the other hand, the FIFO-based method is a kind of approximation of such an FA cache behavior. These judgments cause the difference of 33.3% in average of the maximum absolute errors. The evaluation done by Collins and Tullsen in [4] also showed that their FIFO-based method could identify 88% of conflict misses on the direct-mapped or the 2-way associative cache. From our preliminary evaluation, we observed that the number of FIFO entries is sensitive to the detection accuracy especially for configurations with highly associative caches. Hence, there are no way to completely exclude false judgments in the FIFO-based method. On the contrary, C2Sim accurately models the behavior of cache conflicts based on their definition, and it is robust for highly associative cache structures (such as 20 ways) in modern CPUs.

Considering the actual use cases against performance tuning, the false judgments should be avoided as much as possible to correctly provide the opportunity for cache optimization. For instance, the FIFO-based method correctly reveals the conflicts in covariance while it completely fails in jacobi-2d. If the programmers who perform cache optimization use the wrong target information created by the FIFO-based method, they will never achieve the performance gain from any memory layout optimizations related to line conflicts. On the other hand, our C2Sim can productively reach the precise targets in the performance tuning workflow.

4.4 Reasoning Around Line Conflicts for Performance Tuning

To examine where and why the conflicts occur and to apply these for an actual performance tuning workflow, we pick up three programs (doitgen from PolyBench, 3D-FDTD, Himeno). Figure 3(a) summarizes their cache conflicts detected by C2Sim. Here, we observe conflicts in L1 cache for these programs. For doitgen, we observe conflicts in L2 cache.

	Conflicts [%]		
	L1	L2	L3
doitgen	87.68%	98.44%	0.00%
3D-FDTD (1)	37.45%	0.00%	0.00%
Himeno (2)	92.94%	0.00%	1.76%

(1) FDTD: 128x128x64, timestep=50,
# of mediums (prescribed by array 'id')=10
(2) Himeno Benchmark: OpenMP, C_Dynamic, size=S

(a) Detected cache-line conflicts by C2Sim

```
Memory object-relative view:
malloc[#3] total=336299057 conflictMissPC= 4008a2
  --> malloc[#3] cnt= 308503621, 17620466, 0, originPC= 4008a2
  --> malloc[#1] cnt= 9951731, 107283, 0, originPC= 400898
  --> malloc[#2] cnt= 0, 115952, 0, originPC= 400888 4008c8
  --> Stack(7fff72dde2c4, 4) cnt= 0, 4, 0, originPC= 4008ee
malloc[#1] total=10603920 conflictMissPC= 400898 4008ce
  --> malloc[#3] cnt= 10603920, 0, 0, originPC= 4008a2
malloc[#2] total=115953 conflictMissPC= 400888 4008c8
  --> malloc[#3] cnt= 0, 115953, 0, originPC= 4008a2
================================================
Reason classification view:
          sum          inter-array   intra-array   scalar   unknown
#conflict 347018930:   20894839      326124087     4        0
Ratio                  6.02%         93.98%        0.00%    0.00%
```

(b) A snapshot of observed sources of conflicts (doitgen)

Fig. 3. The detected conflicts and their sources.

To investigate the sources of conflicts further, we analyze the data recorded in the LT-WET structure. Figure 3(b) shows the observed sources of line conflicts during the execution of doitgen. Here, we represent the sources in the following two manners: memory object-relative view and reason classification view. In the memory object-relative view, we track the appearances of conflicts using symbols that represent the memory objects. Here, we see that the malloc[#3] (the thirdly invoked malloc in this execution) causes 336M conflicts at the instruction 0x4008a2. In the following four lines, four of its miss originators are represented with the number of L1, L2, L3 conflicts and their miss originate PCs. Here, we observe that the most significant miss originator is malloc[#3], the same object as the one that causes the miss, and then find the primary reason is intra-array conflict. Similarly, we find the other three originators are caused by inter-array conflict. In the reason classification view, we collect the total number of

intra- and inter-array conflicts for the program execution. From these results, we find that the intra-array conflicts within malloc[#3] are dominant in doitgen.

These information assists us in making strategies for avoiding the unnecessary conflict misses and improving the potential performance of caches. Table 4(a) shows the actual strategies formulated in this paper. While padding is a traditional technique and some existing papers build analytical models to decide the amount of padding [5], the locations to be padded are heuristically determined by hands of expert programmers. Therefore, we propose a workflow that inserts padding to the appropriate arrays suggested through our source analysis mechanism.

Table 4. Cache tuning conducted for doitgen, 3D-FDTD, Himeno.

(a) Strategies for avoiding cache-line conflicts

Tuning strategy	
Opt.1	Intra-array padding insertion
Opt.2	Use of hugetlbfs (2MB page)
Opt.3	Inter-array padding insertion

(b) Cache optimizations in doitgen

Speedup	
Original	1.00
Opt.1	1.19
Opt.1+Opt.2	1.21
Opt.2	1.02

(c) Cache optimizations in 3D-FDTD

Speedup	
Original	1.00
Opt.3	1.32

(d) Scalability for parallel threads and sensitivity to HW prefetch in Himeno

	HW PF	Speedup (**)
1 thread	off	1.62
	on	1.75
16 threads	off	1.50
	on	1.70

(**) **Opt.3** is performed

Table 4(b) shows cache optimizations performed for doitgen and their resultant performance gains. Since the intra-array conflicts within malloc[#3] is the dominant source of conflicts, we insert an extra space within the first dimension of the corresponding 2D array 'C4' in the program. Here, we set 8 elements (64 Bytes) as the amount of intra-array padding to insert an extra space equivalent to one cache-line size. After this optimization (Opt.1), we observe 1.19 times speedup from its original code.

Next, we check whether the conflicts are resolved using C2Sim. The results show that conflicts in L2 still remain although these in L1 are completely eliminated. Here, this phenomenon is derived from the page size used for evaluation. When we use a default 4KB page, the lower 12 bits of memory addresses becomes offsets within the page. Also, for the L1 cache indexing, the lower 12 bits are used. Therefore, all of L1 indexing can be done within a page. However, the L2 and the L3 cache need to use the upper parts of these 12 bits for their indexing, and these are affected by physical address mapping. Since typical linux systems randomize its address space layout through ASLR, the upper parts of the indexes are fragmented. These random index generation makes the effect of intra-array padding diminished. To avoid this, we set 2MB pages through hugetlbfs and control the cache indexing for L2 and L3. After this optimization (Opt.1 + Opt.2), the conflicts within L2 cache are eliminated, and this results in a further speedup.

Here, we note that we cannot achieve such performance improvement if we just adopt hugetlb without intra-array padding (Opt.2).

Then, we shift to the cache optimization for 3D-FDTD. From the result of conflict source analysis using C2Sim, we observe that the conflicts found in L1 are dominated by inter-array conflicts across 7 arrays. Based on this, we insert extra spaces to these arrays. To distribute positions of sets in the L1 cache, we arrange the amount of the padding as $interPad += LineSize \times \lfloor N_{\#sets}/N_{\#arrays} \rfloor$. This means that $M \times 64 \times 9$ bytes padding is inserted at the beginning of Mth array, where $N_{\#sets} = 64, N_{\#arrays} = 7$. From this intra-array padding (Opt.3), we can achieve 1.32 times speedup as shown in Table 4(c).

Next, we examine scalability for multithread executions. Table 4(d) shows the speedup obtained from the original code using an OpenMP version of Himeno. First, we analyze cache miss behaviors for serial and 16-thread execution using C2Sim and observe that both of them are dominated by L1 inter-array conflicts across 7 major arrays[3]. Then, we insert inter-padding by displacing the starting position of each array 64×9 bytes from the adjacent arrays similar to the case of 3D-FDTD. Additionally, to validate feasibility in the actual use cases, we compare the speedup with the configuration that enables the hardware prefetch. From the results, we observe that the performance gain due to the padding is kept even if the number of threads is increased to 16. It is also observed that additional speedup can be obtained when we turn on the HW prefetch. Here, we observe that 1.75x speedup in 1 thread and 1.70x speedup in 16 threads can be achieved compared with their baseline before the padding. From these, we can understand that the optimal padding decision assisted by C2Sim contributes to scalable performance improvement for multithread programs.

We note that the workflow consisting of three strategies (Opt.1 to Opt.3) presented in this Section could be performed automatically by feeding back the dominant source of conflicts to the code generation or runtime parts implemented as a software stack composed of compilers and memory management systems. As a future work, we plan to enhance our C2Sim for a basis of fully automated tuning system.

5 Related Works

C2Sim is a simulation-driven model for detecting cache-line misses to exploit the underlying cache performance. The core part of C2Sim is similar to the algorithm used in cachegrind [1,11], which models SA caches, aligned and unaligned memory accesses and true LRU replacement policy on the top of a dynamic binary translator. In addition to the model found in cachegrind, we implement mechanisms for identifying cache-line conflicts and model more detailed cache structures such as three level caches, physical address translation, and a slice mapping mechanism.

[3] The 16-thread execution might underestimates conflicts in a shared L3 cache because we assume 16 independent L3 caches in the current C2Sim implementation.

CMP$im [7] is a cache simulator implemented using the Pin tool set like ours. While it models details of cache structures across a multi-core CPU, it does not provide any mechanisms to reveal line conflict misses in their original form. On the other hand, C2Sim provides a concurrent dual cache simulation mechanism to accurately identify cache-line conflicts and their sources.

The authors of [19] implement a comprehensive cache simulator that provides cache performance data needed for code optimization. They focus on reuse distance and define conflict miss as follows: If the reuse distance of an access is smaller than the number of cache lines, the resulted miss is regarded as a conflict miss. However, their definition based on reuse distance is a kind of approximation like the FIFO-based method [4,17]. The judgments for conflicts depend on their threshold distance and this leads errors for the detection.

A profiler called DProf presented in [13] uses CPU performance counters to categorize types of cache misses. They attempt to identify line conflict misses (associativity miss in that paper) by finding repeated cycles of the same address in a single associativity set. However, it is not clear how accurate they classify the type of miss using information from hardware performance counters. On the other hand, our C2Sim models line-conflict misses based on theoretical upper bound using FA cache and detect them accurately.

Seshadri et al. proposed a special hardware mechanism called Evicted-Address Filter (EAF) to mitigate cache-line conflicts [18]. They classify line conflict into cache pollution and cache thrashing and attempt to record them on EAF. While their approach can prevent line conflicts to some extent by adjusting cache insertion policy, theirs are hardware-based approach and require modification of hardware.

To the best of our knowledge, this paper is the first one that presents cache-line conflict detection within actual programs using software-based advanced cache simulation techniques. The essential part of this is to reveal detail cache behavior normally invisible from software. Therefore, our simulator is capable of evaluating the impact of different cache organization and strategies like prefetching and replacement policy in addition to cache conflicts focused on this paper.

Padding is a traditional performance optimization technique to avoid cache line conflict misses [5,16]. For inserting pads appropriately, we need to investigate where the extra spaces should be inserted and how much space is good for performance. While some existing papers build analytical models to decide the amount of padding [5,8,14], the locations to be padded are heuristically determined by hands of expert programmers. On the other hand, our C2Sim provides practically accurate sources of conflict and their locations. We believe this dramatically eases the actual performance tuning workflow.

6 Conclusions

In this paper, we have presented a method that reveals cache-line conflicts during the actual execution. Here, we developed a cache-line conflict simulator called C2Sim. C2Sim is capable to simultaneously simulate both ideal fully associative

caches and realistic baseline caches derived from existing architectures. We also proposed a mechanism that enables users to identify where and why line conflict miss occurs. We have shown that cache-line conflict misses can be avoided by padding the appropriate arrays as suggested by our C2Sim analysis. We also showed that these clues manifest themselves in improved execution performance in both serial and parallel executions.

C2Sim is available at https://github.com/YukinoriSato/ExanaPkg as a part of Exana tool kit. We encourage researchers and developers to download it as a basis for productive performance tuning.

Acknowledgments. This work was supported by CREST, Japan Science and Technology Agency.

References

1. Cachegrind. http://valgrind.org/docs/manual/cg-manual.html
2. Himeno benchmark. http://accc.riken.jp/en/supercom/himenobmt/
3. PolyBench. https://sourceforge.net/projects/polybench/
4. Collins, J.D., Tullsen, D.M.: Runtime identification of cache conflict misses: the adaptive miss buffer. ACM Trans. Comput. Syst. **19**(4), 413–439 (2001)
5. Hong, C., et al.: Effective padding of multidimensional arrays to avoid cache conflict misses. In: Proceedings of the 37th ACM Conference on Programming Language Design and Implementation, PLDI 2016, pp. 129–144 (2016)
6. Irazoqui, G., Eisenbarth, T., Sunar, B.: Systematic reverse engineering of cache slice selection in Intel processors. In: 2015 Euromicro Conference on Digital System Design (DSD), pp. 629–636, August 2015
7. Jaleel, A., Cohn, R., Luk, C.-K., Jacob, B.: CMP$im: a pin-based on-the-fly multi-core cache simulator. In: Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MOBS 2008) (2008)
8. Li, Z.: Simultaneous minimization of capacity and conflict misses. J. Comput. Sci. Technol. **22**(4), 497–504 (2007)
9. Luk, C.-K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200 (2005)
10. Minami, T., Hibino, M., Hiraishi, T., Iwashita, T., Nakashima, H.: Automatic parameter tuning of three-dimensional tiled FDTD kernel. In: Daydé, M., Marques, O., Nakajima, K. (eds.) VECPAR 2014. LNCS, vol. 8969, pp. 284–297. Springer, Cham (2015). doi:10.1007/978-3-319-17353-5_24
11. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM Conference on Programming Language Design and Implementation, PLDI 2007, pp. 89–100 (2007)
12. Nikoleris, N., Eklov, D., Hagersten, E.: Extending statistical cache models to support detailed pipeline simulators. In: 2014 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 86–95, March 2014
13. Pesterev, A., Zeldovich, N., Morris, R.T.: Locating cache performance bottlenecks using data profiling. In: Proceedings of the 5th European Conference on Computer Systems, EuroSys 2010, pp. 335–348 (2010)

14. Rivera, G., Tseng, C.-W.: Tiling optimizations for 3D scientific computations. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC 2000 (2000)
15. Satish, N., et al.: Can traditional programming bridge the Ninja performance gap for parallel computing applications? Commun. ACM **58**(5), 77–86 (2015)
16. Sato, S., Sato, Y., Endo, T.: Investigating potential performance benefits of memory layout optimization based on roofline model. In: Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems, SEPS 2015, pp. 50–56 (2015)
17. Sato, Y., Sato, S., Endo, T.: Exana: an execution-driven application analysis tool for assisting productive performance tuning. In: Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems, SEPS 2015, pp. 1–10 (2015)
18. Seshadri, V., et al.: The evicted-address filter: a unified mechanism to address both cache pollution and thrashing. In: 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 355–366 (2012)
19. Tao, J., Karl, W.: Detailed cache simulation for detecting bottleneck, miss reason and optimization potentialities. In: Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools, VALUETOOLS 2006 (2006)