Efficient Cache Simulation for Affine Computations

Wenlei Bao[1(✉)], Prashant Singh Rawat[1], Martin Kong[2], Sriram Krishnamoorthy[3], Louis-Noel Pouchet[4], and P. Sadayappan[1]

[1] The Ohio State University, Columbus, USA
{bao.79,rawat.15,sadayappan.1}@osu.edu
[2] Brookhaven National Laboratory, Upton, USA
mkong@bnl.gov
[3] Pacific Northwest National Laboratory, Richland, USA
sriram@pnnl.gov
[4] Colorado State University, Fort Collins, USA
pouchet@colostate.edu

Abstract. Trace based cache simulation are common techniques in design space exploration. In this paper, we develop an efficient strategy to simulate cache behavior for affine computations. Our framework exploits the regularity of polyhedral programs to implement a cache set partition transformation to parallelize both trace generation and simulation. We demonstrate that our framework accurately models the cache behavior of polyhedral programs while achieving significant improvements in simulation time. Extensive evaluations show that our proposed framework systematically outperforms the time-partition based parallel cache simulation.

1 Introduction

Modern computer architectures leverage memory hierarchies to bridge the speed gap between fast processors and slow memories. At the top of this hierarchy sits the fastest, smaller, and most expensive memory, i.e. registers; at the bottom of the hierarchy, one can find much slower, larger, and cheaper memories (e.g. DRAM, or other permanent media storage such as disks). The intermediate levels of this hierarchy provide temporary storage between two or more levels. This avoids making time-expensive trips to lower memory levels. These intermediate levels are often known as caches, and their main characteristic is to store frequently used data under some pre-determined storage and replacement policy (e.g. LRU, FIFO, etc.).

The behavior of a cache is defined by a number of properties and policies, the most obvious being its memory capacity, the possible locations where a unit of data can be stored, as well as mechanisms for identifying and searching data. Other intricacies of caches include determining when data should be evicted or when it should be committed to a more permanent memory, so as to

L. Rauchwerger (Ed.): LCPC 2017, LNCS 11403, pp. 65–85, 2019.
https://doi.org/10.1007/978-3-030-35225-7_6

keep all levels synchronized [17,27]. In system design, different cache architectures and configurations are thoroughly evaluated to gauge the effectiveness of their use. It is easy to see that the aforementioned cache characteristics increase the complexity of this task. Furthermore, this evaluation is generally performed by trace-driven simulation, which is more flexible and accurate compared to execution-driven cache simulation [25] and modeling approaches [30]. The relevance of cache simulation is also observed in the design of new compiler analyses and transformations. In order for a simulation to be preferred, it must produce the same output as the real program execution while simultaneously being faster or more cost effective. Therefore, fast, reliable and accurate cache simulators are an important tool for engineers and researchers.

Techniques to reduce the complexity of cache simulation have been widely studied in the past. The stack processing algorithm [25] was introduced to reduce the complexity of sequential cache simulation. It was later extended to simulate several cache configurations in a single pass [39], and reduce space complexity by compressing program traces [30].

Trace-driven cache simulation is one of the preferred methods primarily due to the accuracy of its results. However, with the increase in complexity of the cache architectures, trace-driven methods incurs longer simulation times and requires storing large program traces (which store the referenced addresses) to generate accurate results. Therefore, it is imperative to study and develop techniques to overcome these shortcomings. One such technique is the parallel simulation approach [15,26] that exploits the parallelism of current processors to reduce the simulation time.

Previous research efforts in the context of parallel simulation approaches can be divided into two major classes – time-partitioning simulation [15], and set-partitioning simulation [6]. Time-partitioning methods separate the program trace into a number of sequential subtraces of equal length. All the subtraces are used to concurrently simulate cache behavior with identical configuration, and generate partial simulation results. In set-partitioning simulation, a program reference is mapped to a single cache set for a given cache configuration. Each cache set can be simulated independently by a different processor. The parallelism of the set-partitioning approach is therefore limited by the number of sets within the cache configurations.

Time-partitioning parallel simulation can be more efficient than the set-partitioning simulation under certain conditions [6]. However, the accuracy of results for time-partitioning is often lower than set-partitioning because it ignores the initial cache state of each program trace. Subtle algorithmic tweaks are employed to correct/improve the accuracy of the simulation result by performing re-simulation. The added overhead and cost of re-simulation can potentially overcome the benefits from parallelism. Moreover, the time-partitioning scheme involves a preprocessing step that divides the entire program trace into subtraces in order to enable the concurrent simulation. Clearly, the entire trace of the program has to be stored in memory, which may be problematic when the trace size exceeds the system storage.

In this work, we propose a novel parallel cache simulation framework based on set-partitioning for affine programs (where cache set partition based on compile-time analysis in the paper is possible), which achieves up to 100+× speedup against sequential simulation on 64 nodes cluster among 60 evaluations of 10 benchmarks with 6 different cache configurations.

Unlike previous parallel simulation approaches that generate the trace in sequential fashion, our approach also parallelizes the trace production. Moreover, compared to previous set-partitioning approaches [6] that compute the cache set number while performing the trace analysis, our approach organizes the trace by cache set at the very beginning of its generation. It thus avoids costly operations such as insertion and synchronization by maintaining the traces in lexicographic order. To the best of our knowledge, our work is the first to adopt this approach. Experimental evaluations validate the correctness and also demonstrates the performance of our framework.

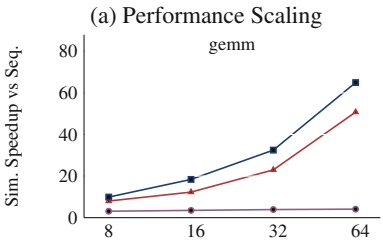
In summary, we make the following contributions:

- We introduce a novel compiler technique that classifies the program references by their targeted cache set. This allows to parallelize the trace generation, and accelerates the overall cache simulation.
- We propose a program transformation that exploits the inherent parallelism exposed by classifying the program references by cache sets, and leverage two standard parallel runtimes, OpenMP and MPI, to increase the cache simulation speed.
- We provide a fully automated tool that, given a C source file containing a polyhedral program region as input, performs the cache set partition transformation and generates code that could conduct the program cache simulation in parallel.
- We perform an extensive evaluation to demonstrate the accuracy of the simulation in terms of cache miss and efficiency in terms of simulation time for our proposed approach.

The rest of the paper is organized as follows: Sect. 2 describes the motivation, Sects. 3 and 4 introduce the background and present the algorithm to perform the cache set partition. Section 5 summarizes our parallel cache simulation framework. Section 6 shows the evaluation results. We conclude the paper with Sect. 7.

2 Motivation

The problem of cache simulation has been extensively studied in the past decades [1,8,16,19,21,25,32,34,38]. Accurate and fast simulation techniques are necessary in order to do extensive architectural space exploration as well as devising new compiler optimization strategies. This problem will become even more important in the current and next generation of computer systems, where



(b) Time and Space cost of Program Trace

Problem Size	Trace (GB)	Time (sec)	Set Trace (GB)	Set Time (sec)
128^3	0.06	1.34	0.001	0.02
256^3	0.5	10.57	0.008	0.17
512^3	4.4	59.18	0.07	0.92
1024^3	36	511.36	0.56	7.99
2048^3	247	2425.18	3.9	37.89

Fig. 1. Motivation example of Gemm

massively parallel processors will effectively have hundreds and thousands of cores. Therefore, good parallel hierarchical cache simulator will play a critical role.

Previous research efforts have demonstrated the benefits of exposing and exploiting parallelism in cache simulations [6,15,31,37]. In particular, parallelization approaches for trace-driven techniques have taken two directions – time-partitioning and set-partitioning. Time-partition approach suffers from inaccuracy and resimulation overhead as previously described. Set-partition approach simulates each cache set independently, does not suffer from the re-simulation problem, and obtains better accuracy. Nevertheless, the achievable speedup is limited to the number of available cache sets. Additionally, most of the previously proposed set-partitioning techniques suffer from the following limitations:

1. Inefficient sequential trace generation phase could dominate the simulation time.
2. Long program runs produce large trace files (potentially in the order of hundreds of GB), which could exceed the storage capacity of a single node.

We now demonstrate the problems in detail with the example of matrix multiplication, *Gemm*, on a real-world cache configuration: a 3-Level cache memory hierarchy with 32 KB 4-way L1, 256 KB 4-way L2 and 8 MB 16-way L3 cache.

Space Complexity of Trace Generation. Table 1b in Fig. 1 lists the space needed to store the trace file for different matrix sizes for *Gemm*. It is easy to observe that even for a problem size of 2048^3, the storage required for the trace file is as huge as 247 GB. Therefore, cache simulation on full-size problems will demand significant storage space, which is impractical.

Time Complexity of Trace Generation. The plot in Fig. 1 illustrates the speedup obtained over the sequential simulation across different number of nodes for problem size of 256^3. *Time-partition+Trace-seq* represents the speedup with time-partition simulation and sequential trace generation, while *Time-partition+Trace-par* shows the speedup of time-partition but with parallel trace

```
1       for (t = 0; t <T; t++)
2         for (i = 1; i <N; i++)
3           for (j = 1; j <N; j++)
4         /* reference c */    /* reference a */     /* reference b */
5       S1:     A[i][j]      =     A[i-1][j]     +      A[i][j-1];
```

Fig. 2. Simplified Seidel-2d benchmark (actual benchmark has 5 read references).

generation. The difference clearly demonstrates that the trace generation phase can dominate the whole simulation time, making the parallel simulation inefficient. Columns *set trace* and *set time* indicate the storage space and time needed if the trace generation phase get parallelized by set-partitioning with 64 nodes. We can observe significant improvements in both space and time consumption. Moreover, the parallelization of trace generation allows the subsequent simulation process to be more efficient compared to previous approaches [6].

Therefore, we propose an efficient parallel cache simulation framework for a class of computationally intensive programs known as affine programs, which automatically transforms the program, and parallelizes the cache simulation along with the trace generation process based on set-partitioning.

3 Program Representation

The compute-intensive kernels of many linear algebra methods, image processing applications [41,43–45], and physics simulations [9,14,40,42] can be expressed as an affine/polyhedral program [2,4,11,14]. Extracting a high performance for such kernels often requires an effective utilization of cache hierarchies.

A property of polyhedral program is that the loop bounds, conditionals, and array indices in the program must be affine functions. The mathematical structures used in this work to represent polyhedral program are: iteration domain, data access relations, and schedule of iterations. The operations on these structures, such as composition and inverse, are the same as [3,5], and not listed here because of the space limitations.

Iteration Domains. Iteration domains capture the set of runtime executions of a statement, using integer sets where the loop bounds are used to constrain the number of points in the set. Each statement S is associated with an iteration vector i_S with one component per surrounding loop, and the values i_S are captured by defining its iteration space $\mathcal{D}_S$. For example the iteration domain of `S1` in Fig. 2 is:

$$\mathcal{D}_{S1} = [T, N] \rightarrow \{S1[t, i, j] : 0 \leq t < T \wedge 1 \leq i < N \wedge 1 \leq j < N\}$$

Data Access Functions. An essential part of our cache simulation framework is the cache analysis based on representing the data accessed by each program iteration. For polyhedral programs, the function that maps a statement instance to the array element being accessed is by definition an affine relation, including

surrounding loop iterators and parameters. The access relation maps an iteration domain to the multidimensional array index being accessed. For example, the function that relates the iterations of $S1$ with the location read in array A for the reference $A[i-1][j]$ in Fig. 2 is:

$$\mathbf{Read}^{A}_{S_1} = \{S_1[t,i,j] \mapsto A[i_2,j_2] : (i_2 = i-1) \wedge (j_2 = j)\}$$

We note **Write** for the write references. Furthermore, one can build the relation that is restricted to the set of iterations of $S1$ by computing $\mathbf{S} = \mathbf{Read}^{\mathrm{A}}_{\mathrm{S1}} \cap \mathcal{D}_{S1}$, that is embed the constraints on the possible values for $S_1[t,i,j]$ directly in the relation.

Finally, we note **ProgRefs** the union of all access relations for the program, **ReadRefs** the union of all read-only access relations for the program, and **WriteRefs** the union of all write-access relations in the program. We have **ProgRefs** = **ReadRefs** ∪ **WriteRefs**.

Program Execution Order. Schedule is used to specify the execution order of statements in program by mapping statement instances in iteration domain to timestamps of iteration space combed with values to indicate orders. As such, statement instances in the iteration domain are executed following the lexicographic ordering $\prec$ of their associated timestamp. $\prec$ is defined as $(a_1,\ldots,a_n) \prec (b_1,\ldots,b_m)$ iff there exists an integer $1 \leq i \leq min(n,m)$ s.t. $(a_1,\ldots,a_{i-1}) = (b_1,\ldots,b_{i-1})$ and $a_i < b_i$.

The program schedule can be denoted by $2d+1$ timestamps, where d is the maximum depth of loop in program [14]. A schedule can be constrained by the iteration domain of its statement, e.g., via $Sched_{S1} \cap \mathcal{D}_{S1}$, and the set of all distinct statement iterations in the program can be built by the union of all schedules constrained by the respective statement iteration domain as **Sched**.

4 Cache Set Partition Analysis

4.1 Cache Access Modeling

The core polyhedral abstractions are obtained from the C code via the PET [36]. In order to model the events corresponding to accessing different cache lines, we must first translate the underlying virtual memory address of each individual array reference into the unique cache line, and the associated set in the cache. For the moment, we will assume that the referenced variable has already been translated to a virtual memory address. Definition 1 defines the steps of the conversion from a given virtual memory address to the associated and accessed cache set.

Definition 1 (Set-associative cache). *A set-associative cache C with associativity K, cache line (i.e., block) size of B bytes, and size n bytes contains S sets, with $S = n/B/K$. A virtual memory address addr maps to a unique line index $L_{id} = floor(addr/B)$, and a line maps to a unique set $S_{L_{id}} = L_{id}\%S$.*

The previous definition essentially assumes that the cache size in bytes (n), the number in bytes of a cache line (B), and associativity degree (K) are given.

We now explain how the translation from a particular array reference to its virtual memory address is performed. The first step is to linearize the access relation. Therefore, for each distinct variable (array or scalar) a vector of fixed dimension sizes is provided at compile-time. The reason for requiring fixed sizes is that the exact virtual memory address must be determined before computing the associated cache set. To complete the linearization, we also require an offset address for each program variable, which in this case, can be either a program parameter or a fixed numerical integer value. If a fixed value is preferred, we can simply estimate the address offsets by taking the declaration order, and computing each array size with the dimension sizes and the floating point precision used. For example, the linearization transformation from a 2D access relation $\mathbf{R}^A$ for array A, with sz the size of A and start_A its starting address can be written as:

$$\textbf{Linearize} = \{[i,j] \mapsto [m] : m = \text{start}_A + i * sz_1 + j\}$$

Then the unique cache line index is given by applying the relation:

$$\textbf{MemToLineId} = \{[m] \mapsto [lid] : lid = \text{floor}(m/B)\}$$

Computing the set to which a cache line maps to is given by the relation:

$$\textbf{LineIdToCacheSet} = \{[lid] \mapsto [cset] : cset = lid \ \% \ S\}$$

Definition 2 (Array to Cache set index). *Given an access function* $\mathbf{R}^A$ *of an array reference* A *with sizes* sz *and starting address* $start_A$*, for a cache as defined in Definition 1. The associated cache line in cache* C *is identified by* **AccessToLine** *as:*

$$\textbf{AccessToLine} = \mathbf{R}^A \circ \textbf{Linearize}(sz, start_A) \circ \textbf{MemToLineId}$$

The composition of the obtained relation with the **LineIdToCacheSet** *relation provides the corresponding cache set in* C *that is referenced by the array and identified by* **AccessToSet** *as:*

$$\textbf{AccessToSet} = \textbf{AccessToLine} \circ \textbf{LineIdToCacheSet}$$

Thus, for every array access within the program, we can determine in a static fashion which cache set it maps to given cache configuration based on above relations.

4.2 Cache Set Partition

In order to distribute and parallelize the trace generation and program simulation, we first need to construct a map from the time space (space of timestamps assigned to each lexical statement) to the accessed cache set. This relation is

easily built by composing the inverse of the program schedule with the composition of the union of access relations with the union of maps that translate the access relation to a specific cache set instance:

$$\textbf{TimeToCacheSet} = \textbf{Sched}^{-1} \circ (\textbf{ProgRefs} \circ \textbf{AccessToSet})$$

where the composition of **ProgRefs** and **AccessToSet** provides the relation from program statements to the cache set index they are mapped to. In a nutshell, the complete equation essentially determines all the timestamps that affect a particular cachet set. The composition is done via the statement instances being accessed. The benefits of having in a closed form all the timestamps mapped to a cache set, is that we can easily determine the subset of statement instances that are associated to the timestamps, and that impact a specific cache set. Obviously, this also allows us to use the schedule map (**Sched**) to generate the necessary array references in the order required by the original program. This step is vital to maintain the program semantics, thereby keeping the original locality and avoiding to insert fake access patterns or remove real ones. Thus, the expression **TimeToCacheSet** calculates the mapping between all assigned timestamps to all different cache sets for affine programs, under the constraints previously discussed.

Hierarchical Cache. To handle multi-level cache hierarchies, we remark that it is easy to build the formulation with the expression above. It can be achieved by editing the cache parameters in **MemToLineId** and **LineIdToCacheSet**, e.g., block size B in **MemToLineId** and number of sets S in **LineIdToCacheSet**. Besides the changes of the formulation for set-partition, the trace analysis algorithm also needs to support multi-level cache simulation, which is shown in the later section. Therefore, an iterative algorithm for program reference behaviors, specializing **TimeToCacheSet** for each cache set value $S_i \in [0, S]$ is built. It can be easily parallelized using either OpenMP or MPI since there are no interactions needed between different threads but a simple accumulation, which compulsory to form the union of cache behaviors such as cache misses for all cache sets.

4.3 Code Generation

The code generation in our framework leverages the result from the previous steps that (a) partition the program statement instances into distinct and individual cache sets; and (b) generate the code that can be execute to conduct the cache simulation in parallel.

There are two phases to achieve the goal of code generation. During the first phase, the union of all statements within an iteration space is scanned using the provided global lexicographic ordering specified by the program schedule, and loop nests in the target program are generated that execute the statement instances in the new lexicographic order. During the second phase, the primary tasks of post-AST processing are (1) Adding parallel/distributed primitives such as OpenMP or MPI for parallel execution; (2) Instrument code to construct

and analyze the trace for cache simulation; (3) Place the reduction code to accumulate the simulation result of each cache set.

Algorithms 1 and 2 detail the code generation steps. Algorithm 1 takes the relation **TimeToCacheSet** together with iteration domains $\mathcal{D}$, and access relations and the original program schedule as input. The overall idea here is that the generated code must contain the original access sequence of the input code; each lexical program statement is decomposed into as many array references as it has; and the original loop nests must be surrounded by an outer parallel loop. This outer loop effectively iterates over all cache sets. In terms of standard loop transformations, this is akin to strip-mining all the original dimensions by the cache set index being accessed.

Line 7 in Algorithm 1 deserves further explanation. The role of **ComputeNewProgramSchedule** is to build the new program schedule from the **TimeToCacheSet** union map. It achieves this by creating a new union map, where the domain is the **TimeToCacheSet** map, wrapped into a set, while the range is a second wrapped map. The second wrapped map has in its domain the same dimensions as the domain of **TimeToCacheSet** with an additional fixed dimension which represents the array reference ID of a specific statement, the **i** argument of the function. The range of the second wrapped map is almost identical to its domain, but where the leading dimension (at position zero) is inserted, and set to the cache set index (which is also the unique dimension in the image of **TimeToCacheSet**). Furthermore, the domains of the second wrapped union map are also properly renamed to prevent fusion among the same points of different statements. Finally, after this map of wrapped union maps to wrapped union maps is computed, we apply a *range* operation to it and return this result.

Algorithm 2 details the post processing steps. It takes the previous generated program as input, and traverse it to enable the instrumentation and proper calls to the parallel runtime of choice. In summary, at this stage we: (1) enable the parallel execution, (2) perform trace generation and analysis and (3) collect the final simulation results by a reduction. Line 1 inserts the parallel primitives for the outer most loop in the transformed program, where the outer most loop is the one to iterate all different cache sets and thus can be easily parallelized. e.g. using #pragma omp for. When using MPI, the described transformations equate to adding a filter to handle the case where many cache sets are assigned to a single process, i.e., we add a filter such as **if (cache_set % comm_size == my_rank)**. Lines 2 to 7 traverse the program to instrument the code for trace production and analysis. Line 8 places the reduction code the collect the final simulation results from parallel processes or distributed nodes. The *AnalyzeTrace* function is responsible of performing the trace analysis and counting the number of cache misses and hits, etc.

Example. Figure 3 presents an example generated by our parallel simulation framework using *Seidel* as the sequential source program. Line 1 to 3 declare the number of sets in the cache configuration and number of nodes available. Line 4 is the loop that decides which cache sets to execute, and depends on the set id and node id match. Line 7 to 10 perform the simulation by construct-

Algorithm 1. Cache Set Partition Code Generation

Input: Program statement iteration domains: $\mathcal{D}^S, s \in S$
Program access relations: $\mathcal{A}^S, s \in S$
TimeToCacheSet relations
Program schedule with cache set dimension: $\theta^S, s \in S$
Output: Cache set partitioned program: $\mathcal{P}$

```
1: for all statements S do
2:    Sort array references of S by lexicographic order and make the write reference
      the last one
3:    for all array references A_i^S in the current lexical statement S do
4:       // Create iteration domains, access relations and schedules for each reference
5:       D^{S,A,i} ← copy D^S, rename set to S_A_i, append fixed dimension and fix to i
6:       A^{S,A,i} ← copy A_i^S, rename the map's domain to S_A_i, append fixed dimen-
         sion to domain of map and fix it to i
7:       θ^{S,A,i} ← ComputeNewProgramSchedule (TimeToCacheSet,i)
8:       // Establish the order among array references of a single statement
9:       Append to the image of θ^{S,A,i} a fixed dimension with value i
10:      // Add computed abstractions to their respective unions
11:      domain ← domain ∪ D^{S,A,i}
12:      access ← access ∪ A^{S,A,i}
13:      schedule ← schedule ∪ θ^{S,A,i}
14:   end for
15: end for
16: P ← codegen(domain,access,schedule)
17: return Generated program P;
```

ing and analyzing the traces on different threads (if using OpenMP) or process ranks (if MPI is preferred). Finally, line 11 is the reduction function to collect the simulation results from all nodes.

Algorithm 2. Post AST processing

Input: Cache set partitioned program: $\mathcal{P}$
Output: Parallel cache simulation program: $\mathcal{P}_S$

```
1: P_S ← Add parallel primitives for outer most loop
2: for all Statements S_i do
3:    for all Array reference R do
4:       T_R ← Construct trace based on reference R
5:       P_S ← Instrument trace analysis code AnalyzeTrace(T_R)
6:    end for
7: end for
8: P_S ← Add parallel reduction code ParReducation to collect results
9: return Parallel cache simulation program: P_S;
```

```
#define SET num_of_cache_sets
#define NODE num_of_nodes
#define DIM SET/NODE
for (c0 = Id*DIM; c0 < (Id+1)*DIM; c0++)//Execute when setId match NodeId
  for (c1 = lb1; c1 < ub1; c1++)
    for (c2 = lb2; c2 < ub2; c2++)
      for (c3 = lb3; c3 < ub3; c3++) {
       AnalyzeTrace(ConstTrace(A[c2-1][c3],c0,c1,c2,c3)); //reference a
       AnalyzeTrace(ConstTrace(A[c2][c3-1],c0,c1,c2,c3)); //reference b
       AnalyzeTrace(ConstTrace(A[c2][c3] , c0,c1,c2,c3));}//reference c
Reduction() {...}; // Reduction code not shown
```

Fig. 3. Example of generated code for Seidel by our framework

5 Parallel Cache Simulation Framework

The overall flow diagram of our set-partition based parallel cache simulation framework is shown in Fig. 4. Our automatic simulation framework works as follows. The input source program is scanned and parsed, the affine computation kernels are extracted and analyzed to construct the relations such as **ProgRefs**, **Sched**, which is performed using ISL. Then the cache set partition analysis and transformation, which is the critical part within the framework, is performed as described in the previous sections. The partition is achieved by the relation **TimeToCacheSet**, which is built upon the cache accessing model and cache set partition formulation, together with the polyhedral analysis. We view this step as *cache set partition transformation*, which reorganizes the programs statements and execution order so that the references accessing the same cache set are grouped together. Next, the code generation algorithm generates the code skeleton of the transformed program, where memory references are grouped based on the calculated cache set number. After that, the post-AST processing algorithm adds the necessary parallel primitives and trace analysis code to generate code for simulation. This is denoted as the *code generation* part in the flow diagram. Finally, during *parallel cache simulation* step, the generated code is compiled and executed in parallel to conduct the trace-driven cache simulation. Thus, the program traces are generated and analyzed in parallel with respect to cache set to produce cache simulation results.

The parallelism of our cache simulation framework comes from set-partition. However, it is better than previous set-partitioning simulation techniques in mainly two aspects. One is the parallelization of trace generation, which exploits

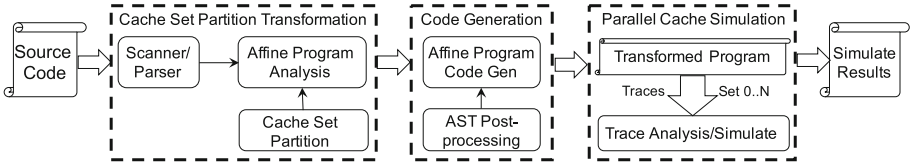


Fig. 4. Parallel trace-driven cache simulation framework

more parallelism within the simulation process and improves the overall performance. The other is the trace analysis process. Previous approaches need to calculate the set number for each trace that involves expensive operations such as trace insertion and synchronization, making the simulation inefficient. In contrast, our approach avoids these operations via the proposed cache set partition transformation, separating the trace based on cache set at source level, which makes the trace analysis much more efficient.

6 Experimental Evaluation

6.1 Experiment Setup

Implementation Details. Our framework takes a sequential C program, cache parameters, array sizes and starting addresses as input. Polyhedral Extraction Tool [36] detects affine regions and extracts the polyhedral model from C source code. ISL [35] is used to perform the cache partition transformation described in previous sections. CLooG [7], a state-of-the-art polyhedral code generator, is used to generate the code based on the algorithm described previously.

Benchmarks. We validate the accuracy and efficiency of our parallel cache simulation framework via the PolyBench/C benchmark suite [29], which is a collection of benchmarks with static control parts that meets our requirements. For the experiments, We select 10 representative benchmarks that are listed in Table 1.

Tools and Setup. To conduct the comparison experiments to validate the performance and correctness with our proposed parallel cache simulation framework, we use DineroIV, a trace-based cache simulator that can handle hierarchical set associative caches, to perform the sequential cache simulation. All experiments are performed on a cluster with a maximum of 64 nodes, each with an Intel Xeon E5640 processor at frequency 2.67 GHz. The programs are all compiled using MPI with MVAPICH2 version 2.1 with -O3 optimization and using one process for each node [18,22–24]. All reported results are the average of 5 runs with single precision used for the benchmarks.

6.2 Experiments Results

We use single- and multi- level set associative caches to validate our framework in both accuracy and efficiency. Note our experiments only show the simulations of most commonly used LRU replacement policy and write allocate write back policy in the evaluation process. However, other replacement policies (FIFO, random, etc.) and write policies (non write-allocate, write through, etc.) are seamlessly handled: their processing is independent from proposed parallel trace generation and simulation.

Single Level Set Associative Cache. We first perform the validation on single level cache with 4 different cache sizes ranging including 4 KB, 8 KB, 16 KB and 32 KB, with block size 64 bytes and 8-way associativity.

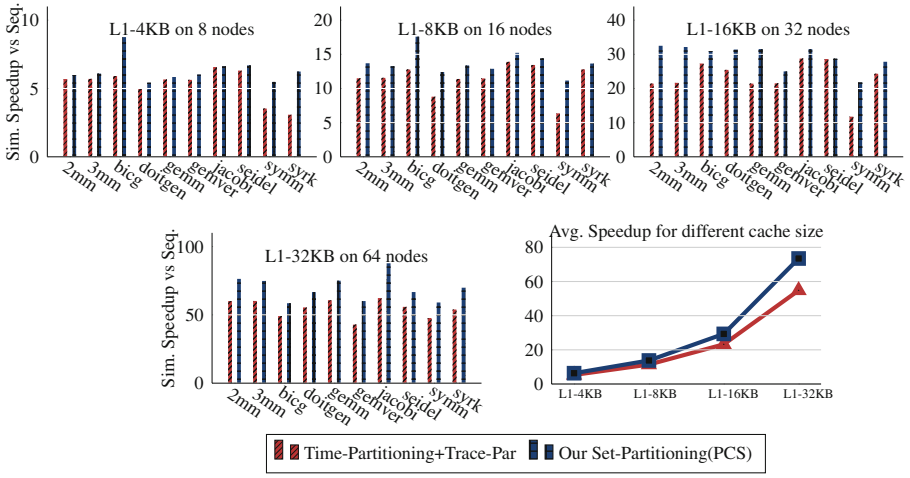


Fig. 5. Summary of simulation speedup for single level cache

Accuracy. The number of cache misses is one of the most important metrics that users want to obtain from the cache simulation to better understand a program's behavior. Table 1 compares the cache miss results for all sizes of single level caches between the sequential simulation with DineroIV and our parallel simulation framework (PCS).

We observe an exact match of the cache misses of the two simulations, which results in an error rate of 0% for all benchmarks across different cache sizes. We can also observe from the table that the cache miss count decreases along with the increasing of cache size until all data can be hold by the cache.

Table 1. Cache misses for single level cache

Sim.	Bench.	Cache configurations				Sim.
		L1-4KB	L1-8KB	L1-16KB	L1-32KB	
Dinero	2 mm	33,846,272	33,709,056	33,709,056	33,708,032	PCS
serial	3 mm	50,769,408	50,563,584	50,563,584	50,562,048	
	bicg	3,146,240	3,146,240	3,146,240	3,146,240	
	doitgen	270,893,056	270,893,056	270,860,288	270,796,800	
	gemm	16,923,136	16,854,528	16,854,528	16,854,016	
	gemver	5,767,936	5,523,312	5,261,165	4,732,719	
	jacobi	12,558,336	12,558,336	8,408,992	8,376,320	
	seidel	6,279,168	6,279,168	2,097,152	2,097,152	
	symm	200,525,957	200,524,323	200,523,289	200,430,141	
	syrk	134,348,800	71,753,728	67,305,472	67,305,472	
Sum.	Error rate	0%	0%	0%	0%	

Efficiency. To further evaluate the performance of our simulation framework, we also compare PCS with *a nearly ideal time-partitioning* based parallel simulation besides the sequential simulation. The time-partition cache simulation divides the whole program trace into multiple, roughly equal sized subtraces, and simulates them in parallel. In our time-partitioning implementation we assume that only one partition requires re-simulation. So the real performance gap between it and PCS could effectively be larger.

Table 2. Summary of cache misses for hierarchical cache

Sim.	Benchmark	Cache configurations							Sim.
		L1	L2	L3-Conf1	L3-Conf2	L3-Conf3	L3-Conf4	L3-Conf5	
Dinero serial	2 mm	33,708,032	707,844	20,480	20,480	20,480	28,399	285,897	PCS
	3 mm	50,562,048	1,061,766	28,672	28,672	28,672	32,784	49,104	
	bicg	3,146,240	1,049,600	1,049,600	1,049,600	1,049,600	1,049,600	1,049,600	
	doitgen	8,387,648	8,384,064	8,371,360	262,144	1,046,424	262,144	285,144	
	gemm	4,332,830	4,329,184	24,800	12,288	12,288	12,288	16,128	
	gemvel	4,732,719	4,722,357	4,721,994	4,722,186	4,722,282	4,323,120	4,722,351	
	jacobi	935,584	934,808	6,873	12,288	12,288	16,128	8,029,984	
	seidel	2,016,000	2,016,000	63,000	63,000	2,016,000	2,016,000	2,016,000	
	symm	200,429,973	192,265,358	49,090	49,090	3,175,278	70,699,265	164,684,202	
	syrk	67,305,472	67,305,472	261,960	1,043,743	67,305,472	67,305,472	67,305,472	
Sum.	Error rate	0%	0%	0%	0%	0%	0%	0%	

Furthermore, normally in time-partitioning based simulation, the whole program trace would need to be first generated and then split into subtraces. This incurs in an inefficient sequential trace generation phase as we already demonstrated in the previous section. It is for this reason that we also combine the time-partitioning scheme with our parallel trace generation. This effectively removes the big performance gap between both schemes. Moreover, here we assume a perfect accuracy of time-partition simulation results even it suffers accuracy problem in reality because of the unknown cache initialization state at the beginning of each subtrace.

Figure 5 illustrates the efficiency of simulation by comparing the speedup of sequential simulation vs. parallel simulation on varying number of nodes. The 4 bar charts in Figure present the simulation speedup between time-partitioning and our framework on different degree of parallelism for all benchmarks across different cache sizes.

We observe that for all the cases, our set-partitioning simulation achieved better speedup compared to time-partitioning. In fact, our set-partitioning simulation constantly outperforms time-partitioning across all the benchmarks. There are several reasons: First of all, the re-simulation phase of time-partitioning approach takes extra cost. In practice, the time cost of the re-simulation phase to correct the simulation results will often make the simulation time much longer than the optimal case we considered here in the experiments. Besides, our set-partitioning approach has better memory efficiency. The memory trace accessed by the program has smaller footprint compared to time-partitioning. This effect is more obvious in hierarchical cache shown later in this section. Line chart in Fig. 5 shows average speedup of all benchmarks for different cache sizes. This is because more parallelism can be achieved for large cache compared to small ones.

Hierarchical Set Associative Cache. We perform similar experiments for multi-level caches for further validation. We consider 5 real world scenario configurations: a 3-Level cache hierarchy with 32 KB 4-way set-associative L1 and

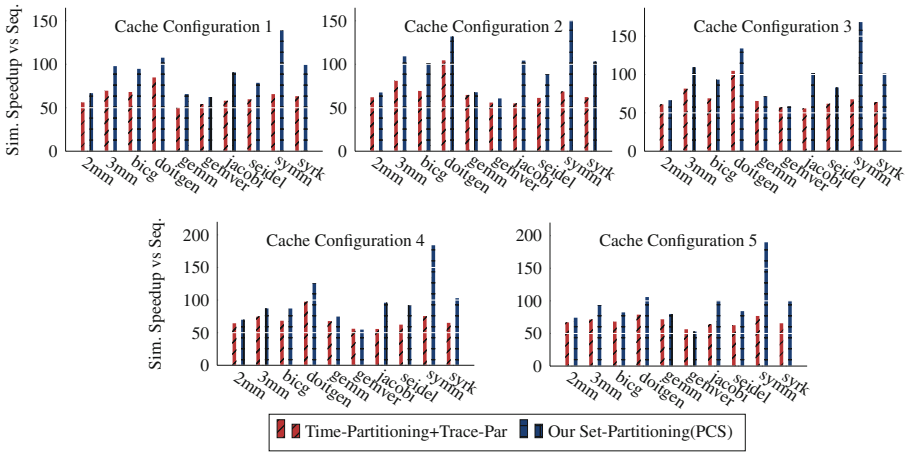


Fig. 6. Summary of simulation speedup for hierarchical cache

256 KB 4-way set-associative L2, and L3 cache with size and associativity reduce by 2 for each configuration start from 8 MB 16-way for Conf1. The block size is 64 Bytes across all levels.

Accuracy. Table 2 compares cache miss count between DineroIV and our parallel simulation framework (PCS) for all evaluated configurations.

Again we observation that *for all cache configurations, our framework produces exactly the same results as DineroIV.* For all benchmarks the cache miss count decreases when moving from the L1 to the L3 cache as expected.

Efficiency. Figure 6 shows the results of performance speedup comparison between time-partitioning and our approach against sequential simulation when using 64 nodes and 1 process for each node (make sure each process has enough computation resources such as cache and memory). We observe that our framework outperforms the time-partitioning approach for all the benchmarks, and across different cache hierarchies.

To illustrate the benefits of our parallel cache simulation framework, we analyze the results of benchmark *symm* in detail. As shown in the chart, *symm* achieves the highest speedup compared to time-partitioning approach among all benchmarks. The underlying reason is that, *symm* uses three matrices of size 512×512, and among the array references, 5 out of 6 of them incur on high-strides. Thus, the non-efficient memory access pattern leads to large cache memory footprints when simulating the full cache. This phenomenon happens again in time-partitioning simulation, as the order of the memory references and cache footprint in trace file remains unaltered. Moreover, the cache set partition transformation changes the memory access order (in simulation) and also the simulation cache footprint. Because every memory reference within each trace is mapping to the same cache set, which has a much smaller cache footprint when

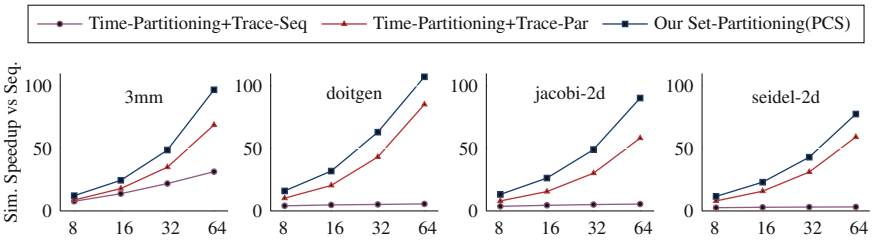


Fig. 7. Summary of performance scaling on cache Conf1

simulating the full cache. Furthermore, the benefits also come from the fact that trace analysis algorithm does not need to spent time on calculating and searching cache set and other related operations in our framework. Therefore, our framework uses a smaller cache footprint during the full cache simulation, and performs much better than the time-partitioning counterpart for benchmark *symm*. Opposing *symm* we have the *gemm* benchmark, which also uses three matrices, but wherein 3 out of 4 matrix references within the innermost loop have stride-1 access. This clearly leads to having rather smaller cache memory footprint compared to *symm*. Thus, the benefits over the time-partitioning on *gemm* are not as large as with *symm*.

Performance Scaling. Figure 7 illustrates the performance scaling of our framework, which is the simulation speedup across different number of nodes (8, 16, 32, 64) with cache configuration *Conf1*. There are three different curves in each subfigure. *Time-partitioning+Trace-Seq* represents time partition parallel simulation with sequential trace generation; *Time-partitioning+Trace-Par* represents time partition parallel simulation with parallel trace generation; *Set-partitioning(PCS)* represents our parallel cache simulation framework. Note we only show 4 benchmarks here because of the space.

It is more than obvious that the simulation with sequential trace generation has limited performance scaling. This demonstrates again the necessity of trace generation parallelization. We also observe that both approaches show strong scaling when increasing the number of nodes. However, our simulation framework outperforms the time-partitioning approach for all the benchmarks by showing a stronger scaling of performance. At this point we also recall that our implementation of the *Time-partitioning+Trace-Par* variant is a nearly ideal and inaccurate simulation, unlike PCS which is as accurate as the serial simulation.

Readers may also observe the super-linear scaling in some benchmarks (e.g. *doitgen*). The reason behind is the cache effect resulting from the different memory hierarchies. With more nodes involved in the computation, the accumulated cache memory (for simulation purposes) also becomes larger, and with larger accumulated cache sizes, more or even all of the working sets can fit into caches and the memory access time reduces dramatically, which causes the extra speedup in addition to that from the actual computation.

7 Related Work

Cache simulation is used to evaluate different cache architectures during new system design. The seminal paper of [25] proposed to use simulation in virtual memory. Their technique computed, in a single pass of the trace file, the miss ratios for all memory capacities, and also introduced notions such as set-refinement and inclusion. However, it was limited to a number of constraints, among of which was a fixed page size. Their work had many applications, in particular, simulation of hierarchical caches.

Due to the constant increase in complexity of cache architectures, a broad range of techniques have been proposed along the years [12,15,28,30,33]. The main difference among the techniques is their cost-efficiency ratio, that is, how much accuracy one is willing to sacrifice in exchange for faster simulation speeds. On one end of the spectrum, parametric analytical models that estimate the number of cache misses are faster and more general available [1,12,13]. On the other end, non-parameterized and less general models combined with trace-driven techniques can be used to produce more accurate simulations, at expense of longer simulation times [8,21,34]. These two classes of work are complementary, and can be used at different stages of the design process.

Compare to cache modeling analysis, simulation still provides a wider coverage of cache architectures and better accuracy. Among all simulation approaches, trace-driven simulation [34], has better accuracy and flexibility. In this context, two directions have been preferred: single pass optimization and trace parallelization strategies. The former one attempts to optimize the simulation in a single sequential pass. This is usually achieved by reducing the trace file size, either by sampling or judicious address selection, and leveraging data structures such as linked lists and trees [8,32] to represent the cache state. Within this research branch, Dinero [10], which is a uniprocessor cache simulator that can handle hierarchical set-associative caches as well as numerous replacement and write policies, thereby characterizing program cache behavior with varying degrees of fidelity.

The second direction aims at partitioning the simulation so that partitions of traces can be executed in parallel [15,19,21,37]. There are two major approaches to exploit the parallelism in cache simulation: time-partitioning and set-partitioning. The idea behind time partitioning is to divide the input program trace into chunks, which can then be simulated in parallel. However, an extra step is necessary to assign the correct cache state between every pair of chunks. Furthermore, depending on the cache configuration and the input program, a number of re-simulation might be necessary and could potentially overcome all parallel benefits, thereby making it even slower than the sequential version. The approach of set partitioning does not require this re-simulation step, since it divides the trace file by the sets addresses by each variable reference. However, the degree of parallelism is limited by the number of sets of cache configuration. Barriga et al. [6] presented a straightforward implementation of cache simulation that exploited set-partitioning. However, their approach included expensive operations such as insertion and synchronization during the trace generation.

Works such as [37], use GPU to exploit the set-partitioning parallelism and simulate multiple cache configurations at one time. Despite utilizing GPUs, their approach still suffers from the inefficiency of processing program traces, specifically, during the address sorting stage.

To the best of our knowledge, in context of trace-driven simulation, all previous works have assumed that the trace generation stage to be inherently sequential. This makes trace-driven cache simulation less efficient as the time spent on generating traces could dominate the simulation time and overcome the benefits achieved via parallelization. Thus, our approach also parallelizes this phase to achieve better efficiency.

Finally, in the general field of simulation, approximate techniques have also been devised. The idea behind this is that results accuracy can be sacrificed in exchange for faster execution times [20]. These techniques have also been adapted for time-parallel cache simulation [19].

8 Conclusion

Exploiting parallelism to accelerate trace-driven cache simulation is a well-studied problem. Previous works have typically focused on two major aspects: (a) the time-partitioning based parallel simulation; and (b) the set-partitioning based approach. These approaches are inefficient when generating and processing large program traces.

In this paper, we propose a novel *parallel* cache simulation framework for *polyhedral programs* to perform accurate, and efficient cache simulation. Compared to previous state-of-the-art works, our approach exploits not only the parallelism in the trace analysis, but also improves the trace generation phase based on cache set partition transformation. Our approach avoids inefficient operations such as trace insertion and synchronization, which are necessary in other set-partitioning methods. We demonstrate that for affine programs, we can achieve better simulation speedup and better memory efficiency compared to time-partition approach. Experimental evaluations validate the accuracy of the proposed framework, showing significant simulation speedup on representative benchmarks against the time-partition parallel simulation.

Acknowledgments. We thank the anonymous referees for the feedback and many suggestions that helped in improving the presentation. This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Awards 66905 and DE-SC0014135, program manager Lucy Nowell, by the U.S. National Science Foundation through awards 1513120 and 1731612, and by computational resources from the Ohio Supercomputer Center. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

References

1. Agarwal, A., Hennessy, J., Horowitz, M.: An analytical cache model. ACM Trans. Comput. Syst. (TOCS) **7**(2), 184–215 (1989)

2. Bao, W., Tavarageri, S., Ozguner, F., Sadayappan, P.: PWCET: power-aware worst case execution time analysis. In: 2014 43rd International Conference on Parallel Processing Workshops, pp. 439–447, September 2014
3. Bao, W.: Power aware WCET analysis (2014)
4. Bao, W., et al.: Static and dynamic frequency scaling on multicore CPUs. ACM Trans. Arch. Code Optim. (TACO) **13**(4), 51:1–51:26 (2016). https://doi.org/10.1145/3011017
5. Bao, W., Krishnamoorthy, S., Pouchet, L.N., Rastello, F., Sadayappan, P.: PolyCheck: dynamic verification of iteration space transformations on affine programs. SIGPLAN Not. **51**(1), 539–554 (2016). https://doi.org/10.1145/2914770.2837656
6. Barriga, L., Ayani, R.: Parallel cache simulation on multiprocessor workstattions. In: 1993 International Conference on Parallel Processing, ICPP 1993, vol. 1, pp. 171–174. IEEE (1993)
7. Bastoul, C.: Generating loops for scanning polyhedra: CLooG users guide. Polyhedron **2**, 10 (2004)
8. Conte, T.M., Hirsch, M.A., Hwu, W.M.: Combining trace sampling with single pass methods for efficient cache simulation. IEEE Trans. Comput. **47**(6), 714–720 (1998)
9. Dundar, M., Kou, Q., Zhang, B., He, Y., Rajwa, B.: Simplicity of kmeans versus deepness of deep learning: a case of unsupervised feature learning with limited data. In: 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), pp. 883–888. IEEE (2015)
10. Edler, J., Hill, M.D.: Dinero IV trace-driven uniprocessor cache simulator (1999). http://www.cs.wisc.edu/markhill
11. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II: multidimensional time. Int. J. Parallel Prog. **21**(6), 389–420 (1992)
12. Ghosh, S., Martonosi, M., Malik, S.: Precise miss analysis for program transformations with caches of arbitrary associativity. In: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VIII, pp. 228–239. ACM, New York (1998). https://doi.org/10.1145/291069.291051
13. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: a compiler framework for analyzing and tuning memory behavior. ACM Trans. Program. Lang. Syst. (TOPLAS) **21**(4), 703–746 (1999)
14. Girbal, S., et al.: Semi-automatic composition of loop transformations. Int. J. Parallel Prog. **34**(3), 261–317 (2006)
15. Heidelberger, P., Stone, H.S.: Parallel trace-driven cache simulation by time partitioning. In: 1990 Proceedings of the Simulation Conference, Winter, pp. 734–737. IEEE (1990)
16. Hill, M.D., Smith, A.J.: Evaluating associativity in CPU caches. IEEE Trans. Comput. **38**(12), 1612–1630 (1989)
17. Hong, C., et al.: Effective padding of multidimensional arrays to avoid cache conflict misses. SIGPLAN Not. **51**(6), 129–144 (2016). https://doi.org/10.1145/2980983.2908123
18. Zhang, J., Lu, X., Panda, D.: High performance MPI library for container-based HPC cloud on InfiniBand clusters, August 2016
19. Kiesling, T.: Approximate time-parallel cache simulation. In: Proceedings of the 36th Conference on Winter Simulation, pp. 345–354. Winter Simulation Conference (2004)

20. Kiesling, T., Pohl, S.: Time-parallel simulation with approximative state matching. In: Proceedings of the Eighteenth Workshop on Parallel and Distributed Simulation, pp. 195–202. ACM (2004)
21. Lauterbach, G.: Accelerating architectural simulation by parallel execution of trace samples. In: 1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences, vol. 1, pp. 205–210. IEEE (1994)
22. Li, M., Lu, X., Hamidouche, K., Zhang, J., Panda, D.K.: Mizan-RMA: accelerating Mizan graph processing framework with MPI RMA. In: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), pp. 42–51, December 2016
23. Li, M., Potluri, S., Hamidouche, K., Jose, J., Panda, D.K.: Efficient and truly passive MPI-3 RMA using InfiniBand atomics. In: Proceedings of the 20th European MPI Users' Group Meeting, EuroMPI 2013, pp. 91–96. ACM, New York (2013). https://doi.org/10.1145/2488551.2488573
24. Li, M., Hamidouche, K., Lu, X., Subramoni, H., Zhang, J., Panda, D.K.: Designing MPI library with on-demand paging (ODP) of InfiniBand: challenges and benefits. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 37:1–37:11. IEEE Press, Piscataway (2016). http://dl.acm.org/citation.cfm?id=3014904.3014954
25. Mattson, R.L., Gecsei, J., Slutz, D.R., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM Syst. J. **9**(2), 78–117 (1970)
26. Nicol, D.M., Greenberg, A.G., Lubachevsky, B.D.: Massively parallel algorithms for trace-driven cache simulations. IEEE Trans. Parallel Distrib. Syst. **5**(8), 849–859 (1994)
27. Patterson, D.A.: Computer Architecture: A Quantitative Approach. Elsevier, Amsterdam (2011)
28. Pieper, J.J., Mellan, A., Paul, J.M., Thomas, D.E., Karim, F.: High level cache simulation for heterogeneous multiprocessors. In: Proceedings of the 41st Annual Design Automation Conference, pp. 287–292. ACM (2004)
29. Pouchet, L.N.: Polybench: the polyhedral benchmark suite (2012). http://www.cs.ucla.edu/pouchet/software/polybench
30. Puzak, T.R.: Analysis of cache replacement-algorithms (1985)
31. Schuff, D.L., Kulkarni, M., Pai, V.S.: Accelerating multicore reuse distance analysis with sampling and parallelization. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, pp. 53–64. ACM, New York (2010). https://doi.org/10.1145/1854273.1854286
32. Sugumar, R.A., Abraham, S.G.: Set-associative cache simulation using generalized binomial trees. ACM Trans. Comput. Syst. (TOCS) **13**(1), 32–56 (1995)
33. Sugumar, R.A.: Multi-configuration simulation algorithms for the evaluation of computer architecture designs (1993)
34. Uhlig, R.A., Mudge, T.N.: Trace-driven memory simulation: a survey. ACM Comput. Surv. (CSUR) **29**(2), 128–170 (1997)
35. Verdoolaege, S.: *isl*: an integer set library for the polyhedral model. In: Fukuda, K., Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 299–302. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15582-6_49
36. Verdoolaege, S., Grosser, T.: Polyhedral extraction tool. In: Second International Workshop on Polyhedral Compilation Techniques (IMPACT 2012), Paris, France (2012)

37. Wan, H., Gao, X., Long, X., Wang, Z.: GCSim: a GPU-based trace-driven simulator for multi-level cache. In: Dou, Y., Gruber, R., Joller, J.M. (eds.) APPT 2009. LNCS, vol. 5737, pp. 177–190. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03644-6_14
38. Wu, M.J., Yeung, D.: Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. ACM Trans. Comput. Syst. **31**(1), 1:1–1:37 (2013). https://doi.org/10.1145/2427631.2427632
39. Wu, Y., Muntz, R.: Stack evaluation of arbitrary set-associative multiprocessor caches. IEEE Trans. Parallel Distrib. Syst. **6**(9), 930–942 (1995)
40. Zhang, B., et al.: Trust from the past: Bayesian personalized ranking based link prediction in knowledge graphs. In: SDM Workshop on Mining Networks and Graphs (MNG 2016) (2016)
41. Zhang, B., Dundar, M., Hasan, M.A.: Bayesian non-exhaustive classification a case study: online name disambiguation using temporal record streams. In: CIKM 2016 Proceedings of the 25th ACM International Conference on Information and Knowledge Management, pp. 1341–1350. ACM (2016)
42. Zhang, B., Dundar, M., Hasan, M.A.: Bayesian non-exhaustive classification for active online name disambiguation. arXiv preprint arXiv:1708.04531 (2017)
43. Zhang, B., Hasan, M.A.: Name disambiguation in anonymized graphs using network embedding. In: The 26th ACM International Conference on Information and Knowledge Management (CIKM 2017) (2017)
44. Zhang, B., Mohammed, N., Dave, V., Hasan, M.A.: Feature selection for classification under anonymity constraint. Trans. Data Priv. **10**, 1–25 (2017)
45. Zhang, B., Saha, T.K., Al Hasan, M.: Name disambiguation from link data in a collaboration graph. In: 2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp. 81–84. IEEE (2014)