PPT-SASMM: Scalable Analytical Shared Memory Model

Predicting the Performance of Multicore Caches from a Single-Threaded Execution Trace

Atanu Barai
Klipsch School of ECE
New Mexico State University
Las Cruces, NM 88003, USA
atanu@nmsu.com

Gopinath Chennupati
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
gchennupati@lanl.gov

Nandakishore Santhi
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
nsanthi@lanl.gov

Abdel-Hameed Badawy*
Klipsch School of ECE
New Mexico State University
Las Cruces, NM 88003, USA
badawy@nmsu.com

Yehia Arafa
Klipsch School of ECE
New Mexico State University
Las Cruces, NM 88003, USA
yarafa@nmsu.com

Stephan Eidenbenz
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
eidenben@lanl.gov

ABSTRACT

Performance modeling of parallel applications on multicore processors remains a challenge in computational co-design due to multicore processors' complex design. Multicores include complex private and shared memory hierarchies. We present a Scalable Analytical Shared Memory Model (SASMM). SASMM can predict the performance of parallel applications running on a multicore. SASMM uses a probabilistic and computationally-efficient method to predict the reuse distance profiles of caches in multicores. SASMM relies on a stochastic, static basic block-level analysis of reuse profiles. The profiles are calculated from the memory traces of applications that run sequentially rather than using multi-threaded traces. The experiments show that our model can predict private L1 cache hit rates with 2.12% and shared L2 cache hit rates with about 1.50% error rate.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**.

KEYWORDS

Performance modeling, Parallel application, Shared cache, Reuse distance analysis, Probabilistic model, LLVM basic block

ACM Reference Format:
Atanu Barai, Gopinath Chennupati, Nandakishore Santhi, Abdel-Hameed Badawy, Yehia Arafa, and Stephan Eidenbenz. 2020. PPT-SASMM: Scalable Analytical Shared Memory Model: Predicting the Performance of Multicore Caches from a Single-Threaded Execution Trace. In *The International Symposium on Memory Systems (MEMSYS 2020),September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3422575.3422806

*Also affiliated with Los Alamos National Laboratory, Los Alamos, NM, USA.

1 INTRODUCTION

With the emergence of Exascale computing and Moore's law coming to a halt, high core counts on multicore processors with complex and large cache hierarchies have become common. Such complicated designs come with several challenges [42], such as the efficient use of available computing cycles, memory delays, and modeling the performance of caches. Designers of parallel applications that run on multicores have to work hard to leverage this extensive computing power. One of the critical factors that determine a parallel application's performance on a multicore processor is the availability of data to the cores. One way to measure an application's data availability is through its cache utilization ability, which directly impacts runtime performance.

Modern processors have shared caches, which significantly impact the performance of an application in the form of data locality and inter-process communication. These factors are both complex to analyze and hardware dependent. Simulation as a modeling tool helps understand and predict applications' behavior and study the impact of the above factors on performance in a multicore configuration. Co-design, which we define as modeling both hardware and software, helps to tune an application's performance. Most of the efforts in co-design have focused on getting simulation data from cycle-accurate dynamic instrumentation tools [17, 24, 26, 45]. However, these simulations require a large number of runs and experimentation with many hardware configurations. Such configurations include variations in cache hierarchies, core counts, and problem sizes, all of which contribute to increasing design space complexity. Using cycle-accurate dynamic simulators to evaluate and predict performance does not scale well. Our solution is to build a scalable simulation model that relies on a detailed cache hierarchy model and application.

In analyzing a cache's performance, *Reuse Distance Analysis* [33] is one of the commonly used techniques. Reuse distance is defined as the number of unique memory references between two references to the same memory reference. For sequential programs, reuse analysis is architecture-independent, whereas for parallel programs that run on multicores, reuse distance dependents on how the memory references of threads interact. Therefore, on multicores, *Concurrent Reuse Distance* (CRD) profiles [19] use a global stack to quantify reuse across thread-interleaved memory references, and thus accounts for data sharing and interaction between threads accessing shared caches. However, CRD profiles are unscalable as the core count increases, and the thread interactions increase; thus, the memory traces get large, which significantly changes the CRD profiles. On the other hand, *Private-stack Reuse Distance* (PRD) profiles depend on how the tasks are scheduled among multiple cores.

In this paper, we introduce the Scalable Analytical Shared Memory Model (SASMM). SASMM relies on the prediction capabilities of the recently open-sourced Performance Prediction Toolkit (PPT) [14]. SASMM is based on reuse distance estimation methods. Our crucial innovation is to include the realistic scenario of caches shared among multiple threads of an application, compared to existing cache models even in the PPT library. SASMM estimates shared and private cache hit rates in a multi-thread and complex cache hierarchy architecture for different applications, which the user can specify. We use a translator based on the *Rose* compiler [31] to get the threaded version of a parallel code written in OpenMP [15]. We develop a compiler-driven technique to identify the threaded programs' basic blocks in measuring the exact probabilities of executing a given basic block of a program. We collect LLVM basic block [30] labeled memory trace from a sequential execution of translated code only once. Using this memory trace, we explore through different scheduling and interleaving strategies of execution to mimic the behavior of multi-threaded programs on shared-memory multicores. These strategies are carried out at the basic block level. We collect a basic block labeled memory trace generated from the translated program's sequential run and apply a probabilistic analytical method to measure both the *PRD* and the *CRD* profiles. Using these profiles, we measure cache hit rates of the applications. We evaluate our approach with the hit rates collected using the Cachegrind tool from Valgrind [34]. The results show that the model accurately predicts cache hit rates compared to hit rates collected using the Cachegrind tool.

2 BACKGROUND

2.1 Execution of Parallel Application: Fork Join Model

OpenMP uses fork-join model for parallel execution of a program. The program begins as a sequential application with a master thread. When the first parallel region construct is encountered, the master thread forks a team of almost identical parallel threads. The forked threads have access to all the variables from the master thread, and those are shared variables. These threads may also have private variables of their own and can identify themselves with unique thread number. When the threads finish executing all the parallel region statements, they synchronize and terminate (join), leaving only the master thread. It is also possible to have nested parallelism where one in the team of threads can fork recursively until it reaches a certain task granularity.

Table 1: Reuse Distance Example

Address	a	b	a	c	b	d	d	a
RD	∞	∞	1	∞	2	∞	0	3

2.2 Reuse Distance Analysis

Reuse distance (D) of a memory address, also known as LRU stack distance, is the number of unique memory references made by a program between two consecutive references to the same address. Note that, when a memory address is referenced for the first time, D's reuse distance is ∞. Reuse profile is the histogram of reuse distances for all memory references of a program. Reuse distance analysis measures the locality [22, 49] of an application, which can be used to predict the cache performance of that application [5, 8, 41] and make cache management policy decisions [23]. For a fully associative cache with capacity C, a memory reference's reuse distance will always trigger a cache miss, if $D \geq C$. Table 1 shows the reuse distance calculation for a sample trace. In the example, 50% of memory references will cause a compulsory cache miss. If we consider that cache size is three, then 13% of all memory references will cause a capacity cache miss. In our work, we calculate the reuse profile at cache line granularity. The addresses we consider to calculate D are cache line addresses.

Reuse distance analysis is robust and architecture-independent for sequential applications. The same reuse profile can be used to determine the performance of different cache sizes. This saves a significant amount of time in cache hit rate analysis as we do not have to collect memory traces for different cache configurations. Many attempts [3, 21, 46] demonstrated the use of memory traces for reuse profile calculations. These approaches use binary instrumentation tools to collect memory traces. The memory traces used in most of these attempts are significant in size and time-consuming to process, thereby unscalable. However, recent attempts from Chennupati et al. [11–13] demonstrated analytical models that scale with a small input run of a program. These attempts help predict the performance of an application on single-threaded programs. In a similar spirit, we model the private and shared cache performance of multicore programs.

2.3 Multicore Reuse Distances

Most of the multicore processors contain both shared and private caches. Although the locality of references of a parallel program in a multicore processor is somewhat architecture-specific, it largely depends on the application's characteristics. The corresponding thread of a core accesses the private cache while the shared cache is accessed through all the cores. Two separate reuse profiles, *Concurrent* and *Private-stack* reuse profiles (CRD and PRD) are used to model shared and private caches [27] respectively. We can interleave memory references from all cores on a single LRU stack to measure concurrent reuse profiles. This interleaving causes different types of interaction: *dilation, overlap, and interception* [47]. Table 2 shows the memory references from two cores. For access of **a** at time 4, CRD is two where its PRD is 1. Here CRD is larger than PRD, which shows *dilation.* On the other hand, data sharing reduces dilation. For the memory reference of **a** at time 9, CRD is three, although there are four memory references between two

Table 2: Concurrent Reuse Distance Example

Time	1	2	3	4	5	6	7	8	9	10
Core C_1	a		b	a	e			d	a	b
Core C_2		c				d	b			
Shared Memory Access	a	c	b	a	e	d	b	d	a	b

consecutive memory references at times 4 and 8. This shows *overlapping* as **d** is accessed by both cores inside reuse interval of **a**. Again for the reference **b** at time 10, the reused data itself is shared. So its CRD is two, which is less than its PRD.

Several recent works have focused on CRD profile and performance prediction of the shared cache [9, 18, 20, 44, 48]. Recently researchers attempted to use an analytical model and sampling to speed up the performance prediction [4, 27, 38–40]. All these models require trace collection from parallel execution of an application for different numbers of threads. On the other hand, our model collects trace once from the sequential run of the application. From that trace, we predict shared cache performance for a different number of threads. This makes our model highly scalable with core counts.

3 PPT-SASMM: SCALABLE ANALYTICAL SHARED MEMORY MODEL

The scalable analytical shared memory model is a parameterized model for the performance prediction of parallel codes. We leverage reuse distance analysis to determine a parallel program's multicore reuse profile that runs on multiple cores. The reuse profiles are later used to determine the hit rates at different cache hierarchies. Figure 1 shows different steps of the analytical shared memory model. Various steps of our model include *a)* translating the OpenMP program to a threaded program, *b)* adding labels for shared variables in the threaded program, *c)* generating a memory trace from basic block labels, *d)* mimicking shared and private memory traces, *e)* estimating private stack and concurrent reuse profiles and hit rates. We describe each of these steps in detail as follows.

3.1 Program Translation

In the first step, we convert the OpenMP application to an intermediate threaded code using OpenMP translator in ROSE [31] compiler. In the translation process, the parallel sections of the original code are transformed into intermediate threaded code. The translation is important in order to track the reuse distances of shared variables. With the high-level OpenMP code, measuring the reuse distances of shared variables is difficult. Therefore, the translated code helps in efficient reuse analysis, thereby the shared cache performance. The threaded version of the code contains XOMP wrapper functions (generated from the Rose compiler), that call GNU OpenMP (GOMP) (when compiled with GCC) library functions. The parallel sections' private variables are translated as local variables in the code's corresponding threaded version. Each thread under execution runs the XOMP wrapper functions, where each thread allocates memory for the local variables. The threaded version of the code's functions receives pointers' structure as a parameter for the shared

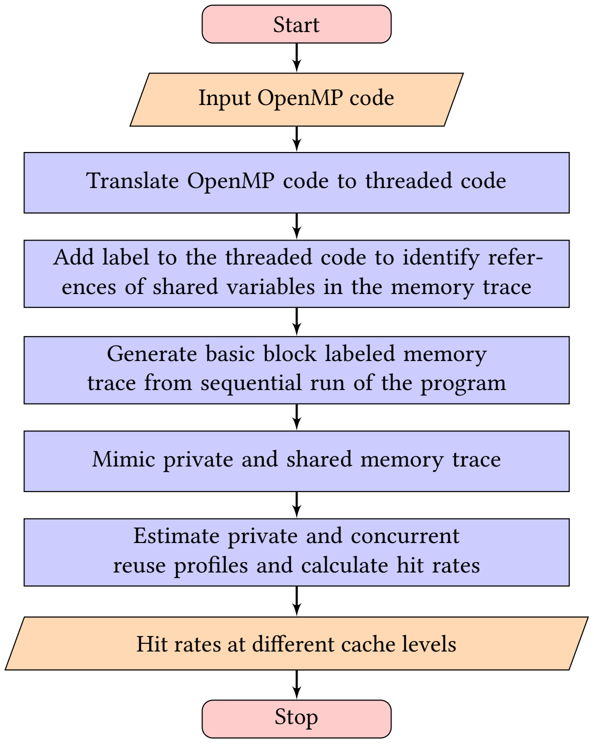

Start
Input OpenMP code
Translate OpenMP code to threaded code
Add label to the threaded code to identify references of shared variables in the memory trace
Generate basic block labeled memory trace from sequential run of the program
Mimic private and shared memory trace
Estimate private and concurrent reuse profiles and calculate hit rates
Hit rates at different cache levels
Stop

Figure 1: Flow Chart of the Scalable Analytical Shared Memory Model (SASMM)

variables. At the beginning of these functions, all the members of those structures are assigned to locally declared pointers. We create separate *labels* for these shared parts of the code, which is where the assignments happen so that the memory trace of the shared variables of the code are grouped in the corresponding basic block labels (described in section 3.2). Figure 3 shows the transformed code of the simple OpenMP code in Figure 2. In the translated code in Figure 3 the static function named *OUT__1__7285* corresponds to the parallel section of the simple OpenMP code in Figure 2. The shared variables are passed to this function using a pointer to the structure named *__out_argv*. We put the assignment statements of shared variables under *shared_var_trace0* label. In the memory trace, all the references under the *shared_var_trace0* label are grouped together.

```
int main()
{
    int i, n;
    int sum = 0;
    n = 500;
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < n; i++)
        sum = sum + i;
}
```

Figure 2: An Example OpenMP Program

```
#include "libxomp.h"

struct OUT__1__7285___data
{
  void *n_p;
  void *sum_p;
}
;
static void OUT__1__7285__(void *__out_argv);

int main(argc,argv)
int argc;
char **argv;
{
  int status = 0;
  XOMP_init(argc,argv);
  int i;
  int n;
  int sum = 0;
  n = 500000;
  struct OUT__1__7285___data __out_argv1__7285__;
  __out_argv1__7285__ . sum_p = ((void *)(&sum));
  __out_argv1__7285__ . n_p = ((void *)(&n));
  XOMP_parallel_start(OUT__1__7285__,&__out_argv1__7285__
     );
  XOMP_parallel_end();
  XOMP_terminate(status);
}

static void OUT__1__7285__(void *__out_argv)
{
shared_var_trace0: {}
  int *n = (int *)(((struct OUT__1__7285___data *)
     __out_argv) -> n_p);
  int *sum = (int *)(((struct OUT__1__7285___data *)
     __out_argv) -> sum_p);
other_trace1: {}
  int _p_i;
  int _p_sum;
  _p_sum = 0;
  long p_index_;
  long p_lower_;
  long p_upper_;
  XOMP_loop_default(0, *n - 1,1,&p_lower_,&p_upper_);
  for (p_index_ = p_lower_; p_index_ <= p_upper_;
     p_index_ += 1) {
    _p_sum = _p_sum + p_index_;
  }
  XOMP_atomic_start();
   *sum =  *sum + _p_sum;
  XOMP_atomic_end();
  XOMP_barrier();
}
```

Figure 3: Transformed OpenMP code using Rose compiler

3.2 Memory Trace Generation for Different Cache Hierarchies

In the second step, we generate LLVM basic block labeled memory trace of the translated threaded program. The LLVM IR of the source code consists of basic blocks, consisting of a single entry and a single exit point. In producing the trace, we execute the translated code sequentially for the parametrized program. We use LLVM based instrumentation to generate the basic block labeled memory trace of the translated program through sequential execution. In this memory trace the i^{th} basic block *(BB_i)* of the labeled trace contains all the memory addresses that are accessed as a result of executing the corresponding straight-line code of *(BB_i)*. For each shared section, marked with a *label*, we gather the corresponding memory references of those shared sections from the trace. Sequentially using this memory trace result, we mimic the memory access behavior of the parallel program and thus generate the private memory trace on each thread under execution.

As OpenMP works within a fork-join model, the parallel section of the OpenMP code is executed at the same time on different cores. Each core has its copy of the parallel section of the code. Note that only the master thread executes the code's sequential part and the corresponding parallel section of the code. We mimic this behavior by making copies of each basic block of the parallel sections' memory references. Our mimicking strategy tries to replicate the memory trace of an OpenMP program on multiple cores. For example, if the parallel program uses 4 cores, we make four copies of a basic block. We then add an *offset* to the memory addresses for each of the cores under execution except the core executing master thread. The basic blocks selected belong to the parallel region of the code. The offset is carried out on all memory references of a parallel region's basic blocks except for the shared variables' memory references. Some basic blocks (*loop iterations*) under the parallel region are executed multiple times. They appear multiple times in the labeled memory trace. After adding offsets in the same way, we distribute the memory references belonging to these basic blocks evenly among all the cores. We choose the offset in such a way that the mimicked memory references do not match with the original memory references produced in the sequential execution. This mimicking strategy helps to show that the memory references belong to different cores.

Algorithm 1 Private Memory Trace Generation

```
procedure $gen\_prvt\_trc(all\_bb, trace, shared\_var\_refs)$
    $each\_core\_trace \leftarrow [[] * num\_cores]$
    $all\_bb\_wins \leftarrow get\_all\_bb\_windows(trace)$
    for $bb_i$ in $all\_bb$ do
        $bb_i\_wins \leftarrow all\_bb\_wins[bb_i]$
        $len\_bb_i\_wins \leftarrow$ len$(bb_i\_wins)$
        if $bb_i$ in $parallel\_bbs$ then
            if $len\_bb_i\_wins == 1$ then
                for $core\_id$ in range $(num\_cores)$ do
                    each_core_trace[core_id] $\leftarrow$
                        $trace\ [bb_i\_wins]$
                end for
            else
                $split\_wins \leftarrow$ array_divide$(bb_i\_wins,$
                    $num\_cores, chunk\_size)$
                for $core\_id$ in range $(num\_cores)$ do
                    each_core_trace[core_id] $\leftarrow$
                        $trace[split\_wins[core\_id]]$
                end for
            end if
        else
            $each\_core\_trace[0] \leftarrow trace[bb_i\_wins]$
        end if
    end for
end procedure
```

The private caches (such as L_1) contain thread-specific execution where each core will have thread-specific memory trace. Therefore, we employ the procedure described in Algorithm 1 to generate private traces for each core, thereby calculating the corresponding reuse profiles and hit-rates. It takes a list of all the basic blocks, the sequential memory trace, and the references belonging to shared variables as input. It finds all instances of each basic block (BB_i) in the memory trace and counts the number of instances. We refer to these instances as windows. If the basic block is in the parallel section with only one instance in the memory trace, we make a copy of that for each core, add offset to the memory references and assign it to each core. If that basic block has multiple instances in the memory trace, then we evenly distribute them to each core. We can perform distribution with chunk size, which is similar to OpenMP static scheduling chunk size. When the basic block is part of the code's sequential region, then we assign all the memory references of that basic block to the core executing the main thread. We find the list of basic blocks using our LLVM based offline code analysis tool.

The original OpenMP execution contains different scheduling strategies (static, dynamic, and guided) to execute the parallel sections. Recording memory traces for such scheduling strategies is cumbersome and inefficient in terms of both time and memory. Therefore, our model in this paper tries to generate a trace similar to the OpenMP scheduled traces. Here, we use the above recorded sequential trace to mimic the interleaving of threads. Our mimicking strategy distributes the corresponding memory threads equally among multiple threads under execution, similar to following static scheduling in OpenMP. We distribute the iterations to the cores according to an adaptive chunk size. In order to further study the effect of scheduling strategies on memory reuse, we propose various interleaving and scheduling strategies, described in section 3.3.

For a shared memory trace, we take the *labeled* memory references from the basic block labeled private traces above. We interleave the memory references of the same basic block from all private traces of the cores sharing that particular memory. We try *round-robin* and *uniform random* scheduling to interleave the memory references. The resultant trace contains all basic blocks' memory trace under sequential execution and interleaved traces of all basic blocks under parallel execution. Thus, the sequence of basic blocks in the mimicked trace is retained from the sequential trace sequence. Similar traces can be generated with binary instrumentation tools such as *Valgrind* [34] and *Pin* [37]. However, we use an LLVM based tool to leverage the conceptual advantage of dealing with simple straight line basic blocks within a program. Valgrind's *Lackey* tool runs the multi-threaded program sequentially per thread, where the threads' interleaving is left to the operating system. Therefore the resultant memory trace happens to be multi-threaded. On the other hand, with Pin, one has to produce a sequential trace and propose interleaving strategies. Nonetheless, we cannot derive a basic block labeled trace from Pin instead of our LLVM instrumentation. We estimate the reuse distances for each reference in the trace, once we have the memory trace that mimics the multicore execution.

Algorithm 2 Interleave memory traces

```
procedure $interleave\_traces(all\_bb, prvt\_mem\_traces)$
    $num\_of\_traces \leftarrow$ len $(prvt\_mem\_traces)$
    $shared\_mem\_trace \leftarrow prvt\_mem\_traces[0]$
    for $bb_i$ in $all\_bb$ do
        if $bb_i$ in $par\_bbs$ then
            for $trace\_id$ in range$(num\_trces)$ do
                $all\_trace\_bb_i\_wins[trace\_id] \leftarrow$
                    $get\_bb_i\_windows(bb\_i, prvt\_mem\_$
                    $traces[trace\_id])$
                $num\_bb_i\_instnc[trace\_id] \leftarrow$
                    len$(all\_trace\_bb_i\_wins[trace\_id])$
            end for
            for $instance$ in range$(num\_bb_i\_instnc[0])$ do
                $ref\_instnc\_i\_all\_traces \leftarrow [[] * num\_trces]$
                for $tr\_id$ in range$(num\_trces)$ do
                    $ref\_instnc\_i\_all\_traces[tr\_id] \leftarrow$
                        $prvt\_mem\_traces[tr\_id][all\_trace\_$
                        $bb_i\_wins[tr\_id][instance]]$
                end for
                $tr\_id \leftarrow 0$
                while $ref\_instnc\_i\_all\_traces \neq []$ do
                    if $strategy == uniform$ then
                        $tr\_id \leftarrow randint(0, num\_trces - 1)$
                    else if $strategy == round\_robin$ then
                        if $tr\_id == num\_trces$ then
                            $tr\_id \leftarrow 0$
                        else
                            $tr\_id += 1$
                        end if
                    end if
                    $interleaved\_bb_i\_trace \leftarrow refs\_instnc\_$
                        $i\_all\_traces[tr\_id].pop(0)$
                end while
            end for
            $shared\_mem\_trace.replace\_bb_i\_refs(interlea$
                $ved\_bb_i\_trace)$
        end if
    end for
end procedure
```

3.3 Interleaving Strategies

To analyze the performance on shared caches (such as L_2), we employ multiple interleaving strategies. This is to mimic the execution strategies of OpenMP constructs over the shared variables. Algorithm 2 takes private traces as inputs and applies our interleaving strategies to generate shared traces for shared memory accesses. The algorithm inputs are a list of all the BB_i and a two-dimensional list of private memory traces of all the various cores under consideration. Note that only the core executing the master thread has memory trace for the sequential section of the code and the trace for the parallel section. We assume that the master thread is being executed in *core 0* without loss of generality. We initiate *shared_mem_trace* with private memory trace of *core 0*. In our next step, we find each basic block's BB_i instances in all the private memory traces and count the number of instances for each trace. Then, we get the memory references and interleave the references for each instance of the private memory traces' basic block. We use either uniform-random or round-robin scheduling to interleave the traces. We replace the n^{th} instance of BB_i in *shared_mem_trace* with corresponding interleaved BB_i instance. The BB_is under sequential execution are not interleaved and remain unchanged. It is thus possible to mimic shared memory traces using this algorithm for any specific cache configuration.

We employ two interleaving strategies: *round-robin* and *uniform-random* (see lines 22–30). We employ these strategies on the sequential trace of a program, which in the end mimics the shared memory trace of a multicore program. For example, when we run a *for loop* for *100* iterations, to mimic the trace of a 4 core execution, we split the 100 executions of each basic block of a *for loop* (note that a for loop, typically contains on the order of 5 basic blocks) into 4 parts, where each part belongs to a single core. We use the 4 part trace to mimic the multicore trace, on which we employ the interleaving strategies. The two interleaving strategies is to experiment with different OpenMP scheduling strategies. In the *round-robin* strategy, for a given basic block, we take the memory reference from each of the four cores, that is core 0, 1, 2, and 3; then we repeat from core 0 to 3 for all the memory references of a basic block. In this way, the shared memory trace for 4 cores is used to calculate the shared reuse profile across 4 cores. In the *uniform-random* strategy, we select a number between 0 and 3 randomly using a uniform distribution. From that trace, we select a memory reference. We repeat this process until all the references from all 4 parts of the trace are finished.

3.4 Calculating Basic Block Probabilities

In the third step, we calculate the probability of executing each basic block from the basic block labeled sequential execution trace of the program. Let us assume that $BB_1, BB_2, ..., BB_j, BB_k, ..., BB_n$ are the basic blocks and any basic block can pass program execution flow to any other basic block. Let us also assume that these basic blocks are executed $N_1, N_2, ..., N_j, N_k, ..., N_n$ number of times respectively. Thus, the apriori probability of executing a basic block $P(BB_i)$ is:

$$P(BB_i) = \frac{N_i}{\sum_{j=0}^{n} N_j} \tag{1}$$

A particular basic block is executed depending on the number of occurrences of a basic block in the memory trace. We get the values of N_i by counting the number of BB_i instances in the memory trace. Note, N_i changes with input size, as a result $P(BB_i)$ also changes. For sequential execution, these probabilities are valid for all levels of caches in the hierarchy. For parallel execution, each core uses a private cache along with shared caches to fetch data. Thus, for parallel execution, these probabilities are valid only for the last level cache. For private caches we calculate BB_i probabilities using Eq. 2 and 3.

$$\underset{j \in serial}{P(BB_{ji})} = \frac{P(BB_i)}{\sum\limits_{k \in serial} P(BB_k) + \sum\limits_{l \in parallel} P(BB_l)} \tag{2}$$

Algorithm 3 $P(D|BB_i)$ Calculation

```
procedure $cond\_reuse\_prof\_BB_i(bb, mem\_trace)$
    $reuse\_dists \leftarrow []$
    $sample\_size \leftarrow x$
    $bb_i\_wins \leftarrow get\_bb_i\_windows(bb\_i, trace)$
    if $len(bb_i\_wins) == 0$ then
        $return\ 0$
    end if
    $sampled\_windows \leftarrow random(bb_i\_wins, sample\_size)$
    for $window$ in $sampled\_windows$ do
        for $idx, mem\_ref$ in enumerate $window$ do
            $rd\_val \leftarrow get\_rd(idx, mem\_ref, mem\_trace)$
            $reuse\_dists.append(rd\_val)$
        end for
    end for
    $unique\_rds, counts \leftarrow unique(reuse\_distances)$
    $p\_rd \leftarrow$ map(lambda $x{:}x/len(reuse\_dists), counts$)
    $reuse\_prof \leftarrow$ zip($uniq\_rds, p\_rd$)
end procedure
```

$$\underset{j \in parallel}{P(BB_{ji})} = \frac{\frac{P(BB_i)}{m}}{\sum\limits_{k \in serial} P(BB_k) + \sum\limits_{l \in parallel} \frac{P(BB_l)}{m}} \tag{3}$$

where, $P(BB_i)$ denotes probability of the basic block under consideration, m denotes the number of cores, $P(BB_k)$ denotes probability of a basic block of sequential region and $P(BB_l)$ denotes probability of a basic block of parallel region. We find the values of $P(BB_i)$, $P(BB_k)$ *and* $P(BB_l)$ from the memory trace of sequential execution. Eq 2 is used for calculating probabilities of basic blocks of sequential region, while Eq 3 is used for basic blocks of parallel region of the program.

3.5 Probabilistic Reuse Profile Estimation

In our next step, we analytically estimate the *Private-stack* and the *Concurrent* reuse profiles of the program (P(D)) from our mimicked private and shared memory traces. The conventional methods of measuring the reuse profile are costly because of the enormous size of the memory traces. We use a technique described in [11], which produces reuse distances at smaller input sizes of a program, and from those reuse distances, we estimate reuse profiles at more massive input sets. We estimate the reuse profile of a program using Eq. 4.

$$\mathrm{P}(D) = \sum_{i=0}^{n(BB)} P\left(BB_i\right) \times P\left(D|BB_i\right) \tag{4}$$

where *n(BB)* is the number of basic blocks, $P(BB_i)$ is the apriori probability of executing a basic block , D is the reuse distance and $P(D|BB_i)$ is the conditional reuse profile of i^{th} basic block.

Algorithm 3 calculates conditional reuse profile $P(D|BB_i)$ of a basic block. It takes the basic block and a mimicked memory trace as input, identifies the basic block's windows in memory trace, randomly selects *sample_size* windows. Typically we randomly select 1% samples of basic block windows. For all memory addresses

Table 3: Applications used to verify our model. †, ∗, and ◇ denote applications from PolyBench/OpenMP [36], Rodinia/OpenMP [10], and PARSEC [6] benchmark suites respectively.

Application	Description	Domain	Input Size	Trace Size	Abbr.
† ADI	Alternating Direction Implicit method for 2D heat diffusion	Stencils	N=512, TSTEPS=2	2.1 GB	**adi**
∗ BFS	Breadth-First Search	Graph Traversal	64K Nodes	1.1 GB	**bfs**
◇ Blackscholes	Black-Scholes partial differential equation	Recognition, Mining and Synthesis	Options=4096, Runs=100	1.7 GB	**blk**
† Convolution-2D	2D Convolution	Stencils	1024	1.5 GB	**c2d**
† Durbin	Yule-Walker equations solver	Linear Algebra	2048	4.2 GB	**dbn**
† Gramschmidt	QR decomposition with modified Gram Schmidt	Linear Algebra	192	3.9 GB	**grm**
† Jacobi	Jacobi Iteration	Stencils	N=1024, Iterations=1024	3.0 GB	**jcb**
† LU	LU decomposition without pivoting	Linear Algebra	256	3.2 GB	**lu**
† 2MM	Two Matrix Multiplication	Linear Algebra	128	967 MB	**2mm**

of each sampled window, we calculate its reuse distance, from which we calculate the corresponding probabilities. This sampling strategy saves significant time in the overall reuse distance calculation. Note that some basic blocks may not be executed at all in the program. In that case, there will be no window of that basic block in the memory trace.

3.6 Hit Rate Estimation

With the probabilistic *Private-stack* and *Concurrent* reuse profiles of each cache level, we measure private and shared cache hit rates using an analytical memory model, a stack distance based cache model (SDCM) [7]. Eq. 5 shows how to measure the hit rate at a given reuse distance ($P(h \mid D)$).

$$P(h \mid D) = \sum_{a=0}^{A-1} \binom{D}{a} \left(\frac{A}{B}\right)^{a} \left(\frac{B-A}{B}\right)^{(D-a)} \tag{5}$$

where D is the reuse distance at cache line granularity, A is the associativity of the cache and B is cache size in terms of number of blocks (which is cache size over cache line size). Typically, Eq. 5 is used for an n-way associative cache. For a direct-mapped cache, probability of hit is defined as

$$P\left(h \mid D\right) = \left(\frac{B-1}{B}\right)^{D} \tag{6}$$

Finally, we calculate approximated unconditional the probability of a hit *P(h)* for the entire program as shown in Eq. 7

$$P(h) = \sum_{i=0}^{N} P(D_i) \times P(h \mid D_i) \tag{7}$$

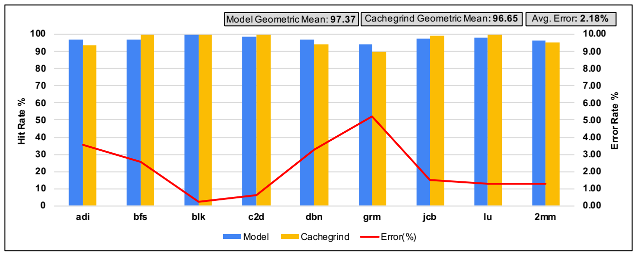
where, $P(D_i)$ is the probability of i^{th} reuse distance (D) in a reuse distribution $Pr(D)$. These hit rates can be further used in runtime prediction of the applications, which is beyond this paper's scope.

4 EXPERIMENTAL RESULTS

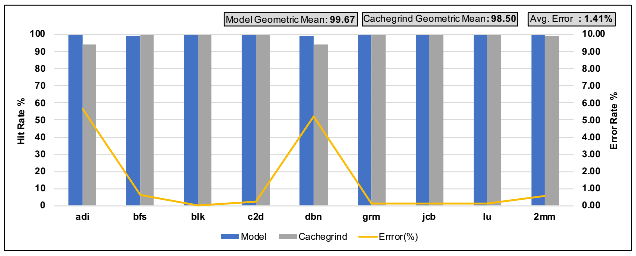
In this section, we validate our model and present the results. Table 3 shows a list of the applications used in the validation. We use nine different applications representing different domains from PolyBench [36], Rodinia [10] and PARSEC [6] benchmark suites. For PolyBench, we use the OpenMP implementation by [25]. We choose these benchmark suites as they are widely used for validating performance models. The generated memory trace sizes are also shown for the input used for each application.

We use the Cachegrind tool within the widely used Valgrind [34] to collect the cache hit rates for different cache configurations. Cachegrind is a dynamic binary analysis tool that performs a trace-driven simulation of a machine's cache as a program executes. The simulated has a split L1 and a unified L2 cache with a write-allocate policy. The L2 cache is inclusive. Cachegrind does not account for interference from the kernel or other processes when it simulates the caches. It is suitable for verifying our model as we try to evaluate the cache performance of the benchmark applications' standalone execution. It also does not account for virtual to physical address mapping. These properties make it an excellent choice to evaluate our method. We consider two cache levels where the L1 cache is private to each core, and L2 is shared among all the cores. The cache configurations used are as follows.

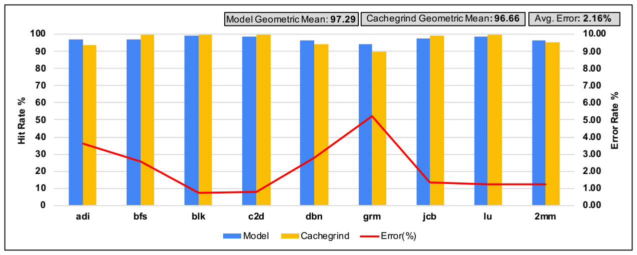
- **L1 D-Cache** *Size:* 8 KB, *Associativity:* 8, *Line Size:* 64 B
- **L2 Cache** *Size:* 128 KB, *Associativity:* 16, *Line Size:* 64 B



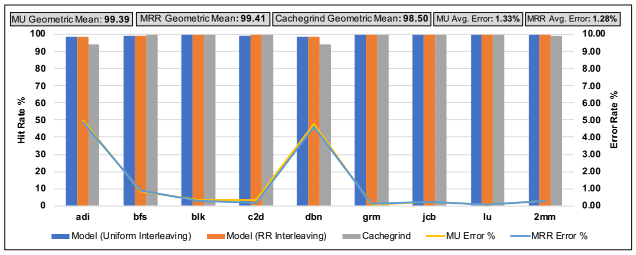
(a) L1 Hit Rates when applications run on 1-core CPU



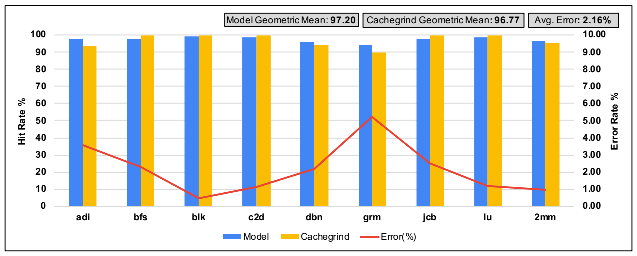
(a) L2 Hit Rates of applications running on single-core CPU (No Interleaving)



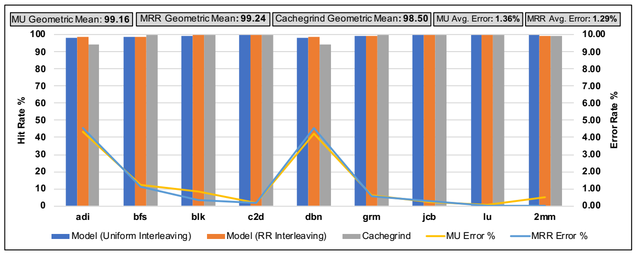
(b) L1 Hit Rates when applications run on 2-core CPU



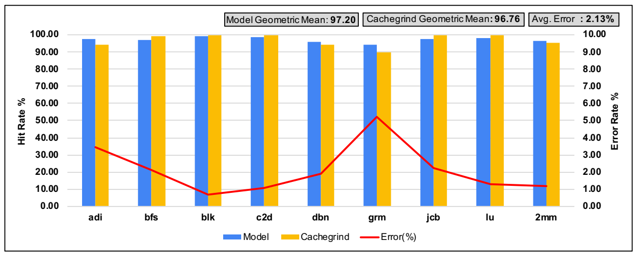
(b) L2 Hit Rates of applications running on 2 core CPU



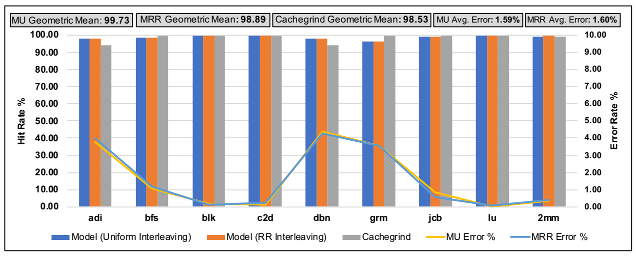
(c) L1 Hit Rates when applications run on 4-core CPU



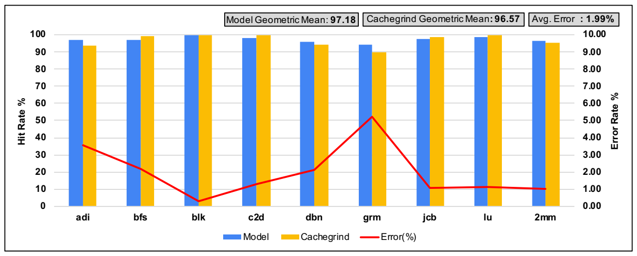
(c) L2 Hit Rates of applications running on 4 core CPU



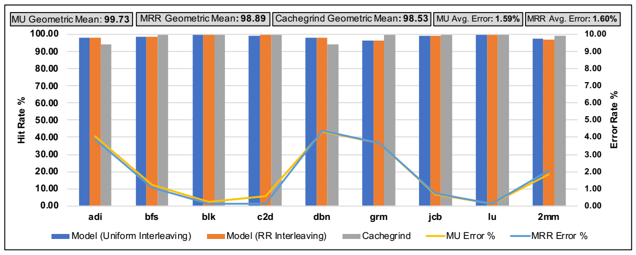
(d) L1 Hit Rates when applications run on 8-core CPU



(d) L2 Hit Rates of applications running on 8 core CPU



(e) L1 Hit Rates when applications run on 16-core CPU



(e) L2 Hit Rates of applications running on 16 core CPU

Figure 4: Hit Rate Comparison on 8KB Private L1-D Cache

Figure 5: Hit Rate Comparison of Applications Running on CPUs with 128KB Shared L2 Cache

Figure 4 shows the comparison of hit rates of private L1 cache for each core configuration. We show the hit rates of the applications running on 1, 2, 4, 8, and 16 cores in Figures 4(a), 4(b), 4(c), 4(d), and 4(e) respectively. We show the geometric mean of hit rates from our model and Cachegrind in the figures. Our model's average error rates are 2.18%, 2.16%, 2.16%, 2.13%, and 1.99% for the core configurations. We compute the average hit rate of private L1 caches obtained using our model. We change the number of threads/cores using *OMP_NUM_THREADS* environment variable when we collect the hit rates using Cachegrind. The results show that our model predicts the hit rates of L1 cache accurately with an overall average error rate of **2.12%**. As we mimic the memory trace of multi-threaded execution from single-threaded execution, we do not consider the effect of cache coherence in our model. Still, the experiments show promising results.

Figure 5 compares the hit rates of shared L2 cache for each core configuration. We show the results for both uniform random and round-robin interleaving. We denote them as **MU** and **MRR** in the figures. Note that, in the results of a single-core configuration shown in Figure 5(a), there is no interleaving. We also show the geometric mean of the hit rates on the L2 cache obtained using our model and Cachegrind. For the single-core configuration, the average error rate is 1.41% for all the applications. Figures 5(b), 5(c), 5(d) and 5(e) show hit rates for core configurations of 2, 4, 8, and 16 respectively. For uniform random interleaving, average error rates are 1.33%, 1.36%, 1.59%, and 1.85%, respectively. These make the overall error rate for uniform interleaving **1.53%**. For round-robin interleaving of memory traces, average error rates are 1.28%, 1.29%, 1.60%, and 1.81% respectively for 2, 4, 8, and 16 core configurations. These make the overall average error rate for round-robin interleaving **1.50%**.

Overall, our error rates appear tolerable. However, we note quite a bit of difference between the different applications that we tested. In particular the *adi*, *grm*, and *dbn* applications give us trouble when predicting L1 cache hit rates almost independent of core count, whereas only *adi*, and *dbn* show higher fault rates on the shared L2 cache at lower core count. Once we move to a larger core count at L2, the model again starts to over-predict hit rates for *grm*.

The over-predictions in the three applications (adi, grm, and dbn) have two reasons depending on the cache (private or shared) model. For private caches, the discrepancies are because our model works with more instances of thread-specific basic block trace instances than the required. Similarly, the shared cache over-predictions are due to the creation of extra basic block traces during interleaving strategies' mimicking behavior. Overall, although we over-predict some of these applications, we observe low error rates, which can be tolerated concerning the ground truth from Cachegrind.

5 RELATED WORKS

Reuse distance [33] analysis has been widely used to predict cache performance [5, 8, 32, 41], make policies for cache management [16, 23, 29] and to predict program locality [3, 22, 27, 49]. Researchers also tried to speed up reuse distance calculation by parallelizing the algorithm [35] and proposing analytical model and sampling techniques [11–13, 43]. Recently, several research works have been done on reuse distance analysis on multicore processors [4, 27, 38–40, 47] and GPUS [1, 2].

Jiang et al. [27] introduced CRD profiles for multicores and provided a probabilistic model to estimate CRDs from the data locality of each thread. They do not consider invalidation for data locality analysis of private caches.

Wu et al. [47] explored PRD and CRD profiles for performance prediction of loop-based parallel programs. They provided a detailed analysis of the effect of core count on PDR and CRD profiles. They also developed a model for predicting PRD and CRD profiles with core count scaling. The predict the CRD profile with about 90% accuracy.

Jasmine et al. [38] proposed a probabilistic method to calculate the CRD profile of threads sharing a cache and derived coherent reuse profile of each thread considering the effect of cache coherence. They derived the concurrent reuse distance (CRD) profile of each thread, sharing the cache with other threads from the thread's private reuse profile.

Schuff et al. [40] explored reuse distance analysis for shared cache accounting inter-core cache sharing. They also studied PRD profiles considering invalidation-based cache-coherence. They further extended their work to accelerate CRD profile measurement by introducing sampling and parallelization [39].

Ding et al. [19] explored theories and techniques to measure program interaction on multicore processors and introduced a new footprint theory. They proposed a trace-based model that computes a set of per-thread metrics. They compute these metrics by single pass over a concurrent execution of a parallel program. Using these metrics, they propose a scalable per-thread data-sharing model. They also propose an irregular thread interleaving model integrated with the data-sharing model.

Kaxiras et al. [28] proposed statistical techniques from epidemiological screening and polygraph testing for coherence communication prediction in shared-Memory multiprocessors.

Almost all of these approaches collect traces at different cache levels from parallel execution of the application. Our approach is different since we collect a trace only once from a sequential execution of the application. This makes our approach very scalable with core count.

6 CONCLUSION

Reuse distance analysis has been a valuable tool for application performance prediction. This paper extends reuse distance analysis to the parallel application domain by accounting for inter-thread interactions for shared caches in a static way. It statically predicts the hit rates of a parallel application on private and shared caches from memory traces of the sequential execution of a single-threaded version of the application. This makes the methodology scalable with core counts and cache sizes. The results show that our model is very accurate for a parallel application's cache hit rate prediction with accuracy ranging from 97.82% to 98.72%. We explore various scheduling strategies of OpenMP with different interleaving strategies using our model. Furthermore, the model takes the cache configuration parameters as input, making it suitable for design space exploration and cache sensitivity analysis.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their feedback. We would also like to thank New Mexico Consortium (NMC) for their continued support and for giving us access to their machines.

The authors would also like to thank Dr. David Newsom for donating several machines to the PEARL laboratory at NMSU. Some of the experiments in this paper were run on the donated machines.

This work is partially supported by Triad National Security, LLC subcontract #581326. Parts of this research used resources provided at the Los Alamos National Laboratory Institutional Computing Program. Computations were run on Darwin, a research computing heterogeneous cluster.

Any opinions, findings, and/or conclusions expressed in this paper do not necessarily represent the DOE or the U.S. Government's views.

REFERENCES

[1] Yehia Arafa, Abdel-Hameed Badawy, Gopinath Chennupati, Atanu Barai, Nandakishore Santhi, and Stephan Eidenbenz. 2020. Fast, Accurate, and Scalable Memory Modeling of GPGPUs Using Reuse Profiles. In *Proceedings of the 34th ACM International Conference on Supercomputing (ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 31, 12 pages. https://doi.org/10.1145/3392717.3392761

[2] Yehia Arafa, Gopinath Chennupati, Atanu Barai, Abdel-Hameed A Badawy, Nandakishore Santhi, and Stephan Eidenbenz. 2019. GPUs Cache Performance Estimation using Reuse Distance Analysis. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. IEEE, IEEE, Piscataway, NJ, USA, 1–8.

[3] Erik Berg and Erik Hagersten. 2004. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004*. IEEE, IEEE, Piscataway, NJ, USA, 20–27.

[4] Erik Berg, Hakan Zeffer, and Erik Hagersten. 2006. A statistical multiprocessor cache model. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Piscataway, NJ, USA, 89–99.

[5] Kristof Beyls and Erik H. D'Hollander. 2001. Reuse Distance as a Metric for Cache Behavior. In *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*. IEEE, Piscataway, NJ, USA, 617–662.

[6] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[7] Mark Brehob and Richard Enbody. 1999. An analytical model of locality and caching. *Tech. Rep. MSU-CSE-99-31* (1999).

[8] Calin Cascaval and David A. Padua. 2003. Estimating Cache Misses and Locality Using Stack Distances. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. ACM, New York, NY, USA, 150–159.

[9] Germán Ceballos, Erik Hagersten, and David Black-Schaffer. 2016. Formalizing Data Locality in Task Parallel Applications. In *Algorithms and Architectures for Parallel Processing*. Springer International Publishing, Cham, 43–61.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[11] Gopinath Chennupati, Nandakishore Santhi, Robert Bird, Sunil Thulasidasan, Abdel-Hameed A. Badawy, Satyajayant Misra, and Stephan Eidenbenz. 2018. A Scalable Analytical Memory Model for CPU Performance Prediction. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Stephen Jarvis, Steven Wright, and Simon Hammond (Eds.). Springer International Publishing, Cham, 114–135.

[12] Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2019. Scalable Performance Prediction of Codes with Memory Hierarchy and Pipelines. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3316480.3325518

[13] Gopinath Chennupati, Nandakishore Santhi, Stephan Eidenbenz, and Sunil Thulasidasan. 2017. An Analytical Memory Hierarchy Model for Performance Prediction. In *Proceedings of the 2017 Winter Simulation Conference (WSC '17)*. IEEE Press, Piscataway, NJ, USA, Article 65, 12 pages.

[14] Gopinath Chennupati, Nanadakishore Santhi, Stephen Eidenbenz, Robert Joseph Zerr, Massimiliano Rosa, Richard James Zamora, Eun Jung Park, Balasubramanya T. Nadiga, Jason Liu, Kishwar Ahmed, and Mohammad Abu Obaida. 2017c. *Performance Prediction Toolkit (PPT)*. Los Alamos National Laboratory (LANL). https://github.com/lanl/PPT.

[15] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[16] Subhasis Das, Tor M. Aamodt, and William J. Dally. 2015. Reuse Distance-Based Probabilistic Cache Replacement. *ACM Trans. Archit. Code Optim.* 12, 4, Article 33 (Oct. 2015), 22 pages. https://doi.org/10.1145/2818374

[17] John D. Davis, James Laudon, and Kunle Olukotun. 2005. Maximizing CMP Throughput with Mediocre Cores. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*. IEEE Computer Society, USA, 51–62.

[18] Sam V. den Steen and Lieven Eeckhout. 2018. Modeling Superscalar Processor Memory-Level Parallelism. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018), 9–12.

[19] Chen Ding and Trishul Chilimbi. 2009. *A Composable Model for Analyzing Locality of Multi-threaded Programs*. Technical Report MSR-TR-2009-107. Microsoft. https://www.microsoft.com/en-us/research/publication/a-composable-model-for-analyzing-locality-of-multi-threaded-programs/

[20] Chen Ding, Xiaoya Xiang, Bin Bao, Hao Luo, Ying-Wei Luo, and Xiao-Lin Wang. 2014. Performance Metrics and Models for Shared Cache. *Journal of Computer Science and Technology* 29, 4 (01 Jul 2014), 692–712.

[21] Chen Ding and Yutao Zhong. 2001. *Reuse Distance Analysis*. Technical Report. University of Rochester, Rochester, NY, USA.

[22] Chen Ding and Yutao Zhong. 2003. Predicting Whole-program Locality Through Reuse Distance Analysis. *SIGPLAN Not.* 38, 5 (2003), 245–257.

[23] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE, Piscataway, NJ, USA, 389–400.

[24] Magnus Ekman and Per Stenstrom. 2003. Performance and power impact of issue-width in chip-multiprocessor cores. In *2003 International Conference on Parallel Processing, 2003. Proceedings.* IEEE, Piscataway, NJ, USA, 359–368. https://doi.org/10.1109/ICPP.2003.1240600

[25] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. IEEE, Piscataway, NJ, USA, 1–10.

[26] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. 2001. Exploring the Design Space of Future CMPs. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*. IEEE Computer Society, USA, 199–210.

[27] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen. 2010. Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors?. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10)*. Springer-Verlag, Berlin, Heidelberg, 264–282.

[28] Stefanos Kaxiras and Cliff Young. 2000. Coherence communication prediction in shared-memory multiprocessors. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*. IEEE, IEEE, Piscataway, NJ, USA, 156–167.

[29] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache replacement based on reuse-distance prediction. In *2007 25th International Conference on Computer Design*. IEEE, NY, USA, 245–250. https://doi.org/10.1109/ICCD.2007.4601909

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–86.

[31] Chunhua Liao, Daniel J. Quinlan, Thomas Panas, and Bronis R. de Supinski. 2010. A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In *Proceedings of the 6th International Conference on Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More (IWOMP'10)*. Springer-Verlag, Berlin, Heidelberg, 15–28.

[32] Rafael K. V. Maeda, Qiong Cai, Jiang Xu, Zhe Wang, and Zhongyuan Tian. 2017. Fast and Accurate Exploration of Multi-level Caches Using Hierarchical Reuse Distance. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, USA, 145–156.

[33] Richard L. Mattson, Jan Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J.* 9, 2 (June 1970), 78–117. https://doi.org/10.1147/sj.92.0078

[34] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (2007), 89–100.

[35] Qingpeng Niu, James Dinan, Qingda Lu, and Ponnuswamy Sadayappan. 2012. PARDA: A Fast Parallel Reuse Distance Analysis Algorithm. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, USA, 1284–1294. https://doi.org/10.1109/IPDPS.2012.117

[36] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench* (2012).

[37] Vijay J. Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. 2004. PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture (WCAE '04)*. Association for Computing Machinery, New York, NY, USA, 22–es. https://doi.org/10.1145/1275571.1275600

[38] Jasmine M. Sabarimuthu and T. G. Venkatesh. 2019. Analytical Derivation of Concurrent Reuse Distance Profile for Multi-Threaded Application Running on Chip Multi-Processor. *IEEE Transactions on Parallel and Distributed Systems* 30, 8 (Aug 2019), 1704–1721.

[39] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/1854273.1854286

[40] Derek L Schuff, Benjamin S Parsons, and Vijay S Pai. 2010. Multicore-aware reuse distance analysis. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, IEEE, Piscataway, NJ, USA, 1–8.

[41] Rathijit Sen and David A. Wood. 2013. Reuse-based Online Models for Caches. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '13)*. ACM, New York, NY, USA, 279–292.

[42] John Shalf, Sudip Dosanjh, and John Morrison. 2011. Exascale Computing Technology Challenges. In *High Performance Computing for Computational Science – VECPAR 2010*, José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25.

[43] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. 2007. Locality Approximation Using Time. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 55–61.

[44] Xudong Shi, Feiqi Su, Jih-Kwon Peir, Ye Xia, and Zhen Yang. 2009. Modeling and Stack Simulation of CMP Cache Capacity and Accessibility. *IEEE Trans. Parallel Distrib. Syst.* 20, 12 (Dec. 2009), 1752–1763.

[45] Guangyu Sun, Christopher J. Hughes, Changkyu Kim, Jishen Zhao, Cong Xu, Yuan Xie, and Yen-Kuang Chen. 2011. Moguls: A Model to Explore the Memory Hierarchy for Bandwidth Improvements. *SIGARCH Comput. Archit. News* 39, 3 (June 2011), 377–388. https://doi.org/10.1145/2024723.2000109

[46] Sam Van den Steen, Stijn Eyerman, Sander De Pestel, Moncef Mechri, Trevor Carlson, David Black-Schaffer, Erik Hagersten, and Lieven Eeckhout. 2016. Analytical processor performance and power modeling using micro-architecture independent characteristics. *IEEE TRANSACTIONS ON COMPUTERS* 65, 12 (2016), 3537–3551. http://dx.doi.org/10.1109/TC.2016.2547387

[47] Meng-Ju Wu and Donald Yeung. 2013. Efficient Reuse Distance Analysis of Multicore Scaling for Loop-Based Parallel Programs. *ACM Trans. Comput. Syst.* 31, 1 (2013), 1:1–1:37.

[48] Yutao Zhong, Steven G. Dropsho, Xipeng Shen, Ahren Studer, and Chen Ding. 2007. Miss Rate Prediction Across Program Inputs and Cache Configurations. *IEEE Trans. Comput.* 56, 3 (March 2007), 328–343.

[49] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program Locality Analysis Using Reuse Distance. *ACM Trans. Program. Lang. Syst.* 31, 6 (2009), 20:1–20:39.