Parallel Loop Locality Analysis for Symbolic Thread Counts

Fangzhou Liu
University of Rochester
USA
fliu14@ur.rochester.edu

Yifan Zhu
University of Rochester
USA
yifanzhu@rochester.edu

Shaotong Sun
University of Rochester
United States
ssun25@u.rochester.edu

Chen Ding
University of Rochester
United States
cding@cs.rochester.edu

Wesley Smith
University of Rochester
USA
wsmith6@cs.rochester.edu

Kaave Seyed Hosseini
University of Rochester
USA
kaave.hosseini@rochester.edu

Abstract

Data movement limits program performance. This bottleneck is more significant in multi-thread programs but more difficult to analyze, especially for multiple thread counts.

For regular loop nests parallelized by OpenMP, this paper presents a new technique that predicts their miss ratio in the shared cache. It uses two statistical models, one for cache sharing and one for data sharing. Both models use a symbolic number of threads, making it trivial to compute the miss ratio of any additional thread count after initial analysis.

The technique is implemented in a tool called PLUSS. When tested on 73 parallel loops used in scientific kernels, image processing and machine learning, PLUSS produces accurate results compared to profiling and reduces the analysis cost by up to two orders of magnitude.

CCS Concepts

• **Software and its engineering** → **Compilers**; Runtime environments; • **Computer systems organization** → *Architectures*.

Keywords

Locality, Concurrent reuse interval, Analytical model, Static analysis, Multi-threaded applications, Data sharing

ACM Reference Format:
Fangzhou Liu, Yifan Zhu, Shaotong Sun, Chen Ding, Wesley Smith, and Kaave Seyed Hosseini. 2024. Parallel Loop Locality Analysis for Symbolic Thread Counts. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3656019.3676948

1 Introduction

On a single processor chip, the number of logical cores increased by 10x in the past decade [34]. When running multi-threaded programs, their cache locality is critical to parallel performance and scalability and has been the target of many past studies. With the trend towards a larger core count continuing, it warrants the development of fast tools to analyze how the number of threads affect cache locality.

Two major approaches can analyze cache performance in modern multicore systems. One is cache simulation [11, 14, 44], which directly collects the cache miss statistics at runtime. However, the data is valid only in one cache configuration. To understand the impact of parallelization on cache performance, multiple rounds of simulation with different core counts and cache sizes are needed. The second is modeling, which derives cache performance, e.g., cache miss ratio, at any cache size by analyzing memory accesses with low overhead [8, 47, 58]. They have to run the target applications multiple times to obtain the cache performance under different thread counts. Moreover, many multi-core cache simulators or profiling tools add synchronization to serialize memory accesses [14, 44], making their speed scale poorly with the thread count.

A more efficient technique would analyze across all cache sizes and all thread counts at the same time, especially for large-scale parallel programs. This paper solves the problem of symbolic parallel locality analysis, where the degree of parallelism is not fixed but as a parameter.

The paper presents a tool called *Parallel Locality Using Static Sampling (PLUSS)*. It targets regular loops parallelized by OpenMP. Instead of profiling, PLUSS uses a recent technique called static sampling, which takes samples of data reuses in a loop without executing the loop [18]. More importantly, its models are symbolic, so there is no need to re-analyze a loop for different thread counts.

The main contributions of this paper are as follows:

- The *Negative Binomial Distribution Model (NBD)*, a new symbolic model of thread interleaving,
- The *relative thread progress uncertainty* assumption, which we establish first in theory and then experimentally,
- The *Racetrack Model*, a symbolic model of data sharing, derived from relative thread progress uncertainty,
- The implementation and experimental evaluation of the accuracy and cost of PLUSS on 73 parallel loops with up to 64 threads, compared with profiling.

In addition, PLUSS can analyze data parallel code beyond those in OpenMP. At the end, the paper demonstrates its use in Rust and compares its parallel implementation in Rust and in C++.

The new solution has four limitations. First, PLUSS assumes that each thread has a similar computation and runs at the same speed *on average*, which is the case for most data parallel code running on multicore CPUs. Second, the analysis is static and does

PACT '24, October 14–16, 2024, Long Beach, CA, USA

ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676948

not handle irregular code, where the memory access depends on the input and changes with the computation and therefore cannot be fully analyzed statically. Third, the new models are symbolic in thread count, but not symbolic in input size. The latter problem is important for static analysis but outside the scope of this paper. Finally, the analysis shows the locality of a parallel program, which determines its data movement, i.e. miss count, but not execution time, which depends on other factors including prefetching. In evaluation, the accuracy is compared with profiling. We do not have a tool to reliably measure the actual number of cache misses.

2 Background

Reuse Intervals (RIs). For each data access, the **Reuse Interval (RI)** is the time between the current and next access to the same data.[1] In this paper, we use logical time, which increments for each access. In "*abccba*" for example, the RI for *a* is 5.

From an RI distribution, the Relational Theory of Locality computes the miss ratio curve for fully associative LRU cache [60]. Extensions have been developed to model the effect of associativity and sub-block granularity [40] and multi-level cache (inclusive or exclusive) [59]. For parallel code, we will show how PLUSS first derives its RI distribution and then its cache miss ratio.

To analyze irregular code, we need to solve two problems. Data reuse can be identified in structured including recursive code using instancewise relational abstract domains developed by Amiranoff et al. [4] as used by Sundararajah and Kulkarni [50] for program optimization. Static sampling [18], which we use in this paper, needs to be extended to irregular code using instancewise analysis. We focus on the effect of parallelism given the result of data reuse analysis. For the rest of the paper, we consider only regular loops.

OpenMP Loop Parallelization. We target regular OpenMP parallel loops [2]. We consider data-parallel loops with no user-inserted synchronization. Synchronization limits parallelism and may cause coherence misses in partitioned caches. While it is important to parallel performance, in this paper we focus on data-intensive loops and the utilization of a shared cache, which is a major factor determining the magnitude of data movement. Past work has also exclusively focused on OpenMP parallel loops without synchronization [36, 53].

When a loop is parallelized by OpenMP directives, its iterations are divided into fixed-size *chunks*. The scheduling algorithm behind the OpenMP `schedule` clause determines the chunk-to-thread mapping. PLUSS models this mapping behavior of OpenMP schedulers (details are omitted for brevity). We support two OpenMP `schedule` clauses: `static` and `dynamic`. In *static scheduling*, chunks are distributed to threads round-robin; in *dynamic scheduling*, chunks are distributed to threads whenever they are idle.

[1]The reuse interval has been called the inter-reference interval (iri) in the working set theory [19], inter-reference gap in LIRS [32], reuse distance in StatCache and StatStack [21], and reuse time in cache sharing models [57].

[2]Loop controlled by a loop-associated OpenMP directive (also called associated loop in OpenMP).

3 Parallel Locality Analysis

This section first describes the problems of parallel locality analysis and then presents two new models and the PLUSS system design and optimization.

3.1 Effects of Parallelism on Locality

Locality in a program comes from data reuse. A reuse interval is the time between the use and the reuse. Parallelization has two effects on data reuses: dilation, where a reuse interval is lengthened due to co-run threads accessing other data; and intercepts, where another thread accesses the same datum, and a reuse interval is split into two [55]. Dialtion is reuse expanding, while an intercept is reuse splitting. The former is the effect of cache sharing, which is interfering and always negative. The latter is the effect of data sharing, can be collaborative and can improve locality.

To simplify our presentation, we name the thread-local reuse interval *private reuse interval (PRI)* and the reuse interval after thread interleaving *concurrent reuse interval (CRI)*.[3] Figure 1 shows the two effects in a two-thread execution example.

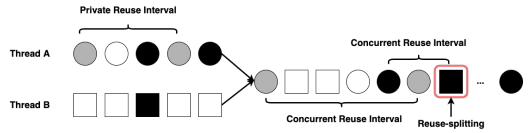

Figure 1: Two example effects of parallelism on reuses. Circles and squares represent memory accesses of two threads A and B. Those in the same color (black or gray) represent data reuses. The CRI of the gray circle is affected by interleaving, while the CRI of the black circle is affected by data sharing.

PLUSS consists of two new models: the negative binomial distribution (NBD) model to handle the dilation effect due to cache sharing and the racetrack model to handle intercepts due to data sharing. Table 1 shows modeling based on data sharing and no data sharing. In each case, an RI is classified in two types as short or long, and the modeling is based on the RI type.

Table 1: Modeling based on RI and Data Sharing

	Short RI	Long RI
Data Sharing	*NB Distribution*	*Racetrack Model*
No Sharing	*NB Distribution*	*Scale by thread count*

3.2 Intra-thread Reuse Model

Intra-thread reuses are those reuse pairs whose reuse source and sink come from the same thread. The RI of such a reuse window after thread interleaving will always be greater than or equal to its *PRI* – such cases are the "reuse-expanding" effect discussed previously.

[3]These names are adapted from private reuse distance (PRD) and concurrent reuse distance (CRD) used by Wu and Yeung [55] and others (See Section 6).

Assumption 1 (Statistical Uniform Interleaving). *Suppose there are T threads. At each step during execution, a thread is chosen to execute uniformly (with equal probability) at random among these T threads.*

With our statistical assumption of the thread interleaving, each access in the reuse window can then be viewed as the result of an independent and identically distributed (i.i.d.) Bernoulli trial with two canonical states: *SUCCESS* and *FAILURE*. Success means this access comes from thread t, while failure means it is an access in a thread t', $t \neq t'$. The probability that the trial is in a success state is the probability thread t is chosen, which is $\frac{1}{T}$. The *CRI* of a reuse window then can be viewed as a random variable representing the number of independent Bernoulli experiments given *PRI* number of successes. In probability theory and statistics, such random variables are said to follow the negative binomial distribution with parameters r and p, where r is the number of successes and p is the probability of success. In our problem, r is *PRI* and p is $\frac{1}{T}$.

Equation 1 presents the probability mass function of Y, $Y \sim NB(r, p)$ if we define it as the *CRI* of each reuse window. It computes the probability of $Y = y$ given its $PRI = r$.

$$P(Y = y) = \binom{y-1}{r-1}(\frac{1}{T})^r(1-\frac{1}{T})^{y-r} \tag{1}$$

Distributing every reuse interval can be time-consuming. Finding a good estimation of *CRI* could be beneficial. With this estimation, *CRI* can be computed to a value instead of a distribution, and hence, improve the performance of the model. In the next theorem, we derive a lower bound of *PRI*. The *CRI* of a reuse pair is concentrated around its expectation if its *PRI* is longer than this bound.

Theorem 3.1 (Chernoff-Hoeffding Bound of CRI). *Let c_1, c_2 are two constants and $0.5 < c_1 < 1$ and $c_2 > 1$. Given a private reuse interval $PRI = r$, after the thread interleaving, it is highly likely (e.g., ≥ 0.999 probability) that its CRI is concentrated near its expectation $\mu = T \cdot r$, written as $P(c_1 \cdot T \cdot r \leq CRI \leq c_2 \cdot T \cdot r) \geq 1 - \epsilon$ if the PRI satisfies the condition:*

$$r > \max(\frac{2\ln\frac{1}{\epsilon}}{c_2 \cdot (1/c_2 - 1)^2}, \frac{3\ln\frac{1}{\epsilon}}{c_1 \cdot (1/c_1 - 1)^2}) \tag{2}$$

Where ϵ is a small number.

This bound conveys that if the $PRI = r$ is long enough, its CRI distribution concentrates on its expectation, $T \cdot r$. Interestingly, if threads are interleaved in a round-robin fashion, their *PRI* will be scaled by the thread count, T, making its CRI also $T \cdot r$. This implies the following: if a PRI is short, it is important to use the negative binomial distribution to calculate a CRI distribution; otherwise, CRI is $PRI \cdot T$, i.e. statistical uniform interleaving is effectively round-robin.

3.3 The Racetrack Model

The racetrack model has an intuitive explanation. Imagine array data as a circular racetrack, and threads are runners with two properties:

- *Same average speed*, each runner may go fast or slow but all have the same average speed, and
- *Random gaps*, at any given time in the steadystate, the distance between runners is random, regardless of their relative positions at the beginning.

The first property is the result of Assumption 1. Next we show that the second property can be derived from the first. Mathematically, both properties come from just one assumption. In this section, we derive these properties mathematically. Later we will verify experimentally (the second property in Section 3.4 and the overall model in evaluation).

The second property may seem too simple given that a wide variety of factors including operating system design, hardware optimization, memory hierarchy structure, algorithm behavior, and others affect the rate of progress in different threads in a parallel execution. These rates can vary greatly in practice. Next, we show that the property is a logical consequence, if we assume statistical uniform interleaving (Assumption 1).

First, Assumption 1 determines the *Thread progress probability* as follows: At each timestep during program execution, each thread independently generates a memory access with probability p and does nothing with probability $(1 - p)$ for some p.

Assume that two threads traverse an m-element array repeatedly, i.e., $a_1a_2 \ldots a_m a_1 a_2 \ldots a_m \ldots$ We derive the second property from Assumption 1.

A racetrack is a spatial interpretation of the thread execution. Picture the m data as slabs lined up in a circular track. As it executes, a thread runs loops on the racetrack. At each time point, the thread accesses a data element. The thread progress is given by its position on the racetrack.

Consider a Markov chain where states represent the gap between the position of two threads (measured by the number of data elements). There are m states labeled $0 \ldots (m-1)$. Assumption 1 yields the random walk-style chain shown in Figure 2.

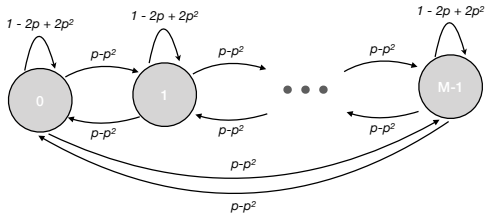

Figure 2: The Markov chain from our progress assumption

Each state represents a possible gap between two threads. A transition happens only when one thread makes a data access, and the other does not. There are two symmetrical cases. The probability of the gap expanding and shrinking (by one) is $p(1-p) = p - p^2$ in both cases. Hence, the probability of staying at the same state is $1 - 2p(1-p) = 1 - 2p + 2p^2$. The transition matrix P for this chain is detailed in Equation 3. This lets us prove the following theorem.

$$P = \begin{bmatrix} 1-2p+2p^2 & p-p^2 & \cdots & 0 & p-p^2 \\ p-p^2 & 1-2p+2p^2 & \cdots & 0 & 0 \\ \\ 0 & p-p^2 & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & p-p^2 & 0 \\ \\ \vdots & \vdots & \cdots & 1-2p+2p^2 & p-p^2 \\ p-p^2 & 0 & \cdots & p-p^2 & 1-2p+2p^2 \end{bmatrix} \quad (3)$$

Theorem 3.2 (Racetrack Model Stationary Distribution). *The stationary distribution of the Markov chain representing the relative positions of two threads under Assumption 1 is uniform.*

Proof. From the Markov chain transition matrix in Equation 3, we can find this chain's stationary distribution by solving $P\pi = \pi$, where π is the stationary distribution. Since 1 is an eigenvalue of this matrix, with corresponding eigenvector $v = (1, 1, ..., 1)$, the stationary distribution $\pi = \frac{1}{m}v$ is uniform. □

This result has a simple high-level explanation: if two threads are processing the same sequence of memory accesses, given a position for thread 1, *thread 2 is equally likely to be anywhere in that sequence.*

Remark. Notice that the modulo effect is crucial here. The Markov chain is connected in a ring structure due to modulo. Otherwise, one cannot obtain uniform distribution of the differences among threads. This Markov Chain model, captures the fact the thread may have equal chance to progress. So the next distances among threads are dependent of current state, where possible changes of distances on each step are specified by transition probability. The uniformity of the distance between threads is resulted from the cyclic chain.

For this to be useful for practical length traces, we must show that the mixing time of the markov chain is not exponential in data size. In other words, we must show that we do not need an exponential length trace for our stationary distribution to be an effective model.

Because this chain is reversible and ergodic, we can apply the well known theoretical result [35] that

$$t_{mix}(\epsilon) \leq \frac{1}{\gamma_*} log(\frac{1}{\pi_* \epsilon})$$

where γ_* is the chain's *spectral gap*, or the difference between the first and second largest absolute values among the transition matrix eigenvalues, and π_* is the smallest element of the chain's stationary distribution. Here, $\pi_* = \frac{1}{m}$.

What remains is to compute γ_*. To do this, we will relate transition matrix P to the corresponding non-lazy random walk Markov chain, which moves left or right on the same state space with probability $\frac{1}{2}$ and has no chance to remain in the same state. Let the transition matrix of this random walk be R. Then

$$P = (1 - 2p + 2p^2) \cdot I + 2(p - p^2) \cdot R$$

Because the identity matrix has only eigenvalue 1, we can express the largest and second largest eigenvalues as sums of the eigenvalues of the previous decomposition:

$$\begin{aligned} \gamma_* &= \lambda_1^P - \lambda_2^P \\ &= (\lambda_1^{(1-2p+2p^2)\cdot I} + \lambda_1^{2(p-p^2)\cdot R}) - (\lambda_2^{(1-2p+2p^2)\cdot I} + \lambda_2^{2(p-p^2)\cdot R}) \end{aligned}$$

Simplifying by cancelling eigenvalues of I:

$$\gamma_* = (\lambda_1^R - \lambda_2^R) \cdot 2(p - p^2)$$

So all we still need are the eigenvalues of R to compute mixing time. For this simple random walk, the following is well known:

$$\lambda_1^R = 1, \quad \lambda_2^R = \cos(\frac{2\pi}{m})$$

where m is the number of states in the chain as before. It follows that:

$$\gamma_* = (1 - \cos(\frac{2\pi}{m})) \cdot 2(p - p^2)$$

Expanding the cosine power series:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} ...$$

Plugging in $\frac{2\pi}{m}$:

$$\gamma_* = (c_1 \cdot \frac{1}{m^2} - c_2 \cdot \frac{1}{m^4} ...) \cdot 2(p - p^2)$$

For constants c_1, c_2.... Lastly, we remove all insignificant terms (preserving our upper bound) and plug back into our original form for mixing time:

Theorem 3.3 (Racetrack Model Mixing Time). *The Markov chain described by transition matrix P for m elements mixes as follows, c is a constant:*

$$t_{mix}(\epsilon) \leq c \cdot m^2 \cdot log(\frac{m}{\epsilon})$$

Next we will show how the racetrack model can be applied to understand the effect of parallelism on reuse. To understand how reuses are "split" under the effects of the previous Markov model, we must first make an assumption about data sharing.

Assumption 2 (Data sharing). *All threads access the same shared data and access it in the same sequence.*

For an example, consider parallel matrix multiplication where each thread computes on different rows of A, C matrices but reuses the B matrix, traversing it in an identical order. The A, C matrices are not shared, but B is.

Each thread may access the shared data repeatedly, e.g. if a thread multiplies on multiple rows. If the size of shared data is non-trivial, during a single traversal by a thread, every *other* thread will access most of the shared data only once.

Combining the two assumptions, interleaving and data sharing, with Theorem 3.2, we can now see that the effect of parallelism reduces to dropping points randomly on a line segment and observing the lengths of the partitioned result as (cross-thread) reuse intervals. For T threads and a sequential reuse interval r, the racetrack model computes the distribution of RIs resulting from randomly sampling $T - 1$ points in the interval $[0, r]$ and partitioning the interval using those points.

3.3.1 Randomly sampling a line segment. Let X be the length of a segment resulting from randomly generating n points from [0,1] (the line segment is normalized to 1) and using those points to partition this normalized line segment. Without loss of generality, assume that X is the length of the leftmost segment, i.e., the segment between 0 and the smallest sampled value. This assumption is validated by the fact that the random variables corresponding

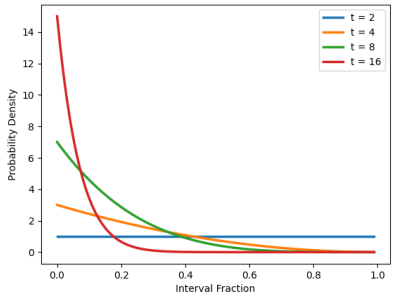

Figure 3: The distributions of line segment lengths resulting from the racetrack model for varying thread count

to each segment will have identical probability density functions, which can be proved by considering this problem as the case of partitioning a unit circle with $n+1$ points, then randomly choosing one of the points to "split" into the line segment.

The cumulative distribution function for X is written as follows:

$$F_X(x) = P(X \leq x)$$

Now, note that $P(X \leq x)$ is the complement of the probability that all points have values larger than X. Mathematically:

$$F_X(x) = 1 - (1-x)^n$$

We then differentiate and apply the chain rule to derive the probability density function $f_X(x)$, our desired result:

$$f_X(x) = \frac{d}{dx}F_X(x) = -1 \cdot (-1 \cdot n \cdot (1-x)^{n-1}) = n \cdot (1-x)^{n-1}$$

So, the probability density function $f_X(x)$ of the distribution of lengths resulting from partitioning [0,1] into $n+1$ segments with n randomly chosen points is $f_X(x) = n \cdot (1-x)^{n-1}$.

Theorem 3.4 (The Racetrack Model). *Let the assumptions from the previous sections hold true, and let there be T threads. Then the probability density function for a fraction x of a thread-local reuse interval that remains under splitting by accesses from other threads is*

$$f_X(x) = (T-1) \cdot (1-x)^{T-2}$$

As mentioned at the start of this section, intuitively, the model states that threads (as runners) are randomly distributed over a racetrack. Figure 3 visually demonstrates the distributions of segment lengths that Theorem 3.4 yields for thread counts of 2, 4, 8, and 16.

Applicability. The racetrack model consists of two properties: thread relative positioning (Theorem 3.2) and the CRI distribution (Theorem 3.4). The first depends only on Assumption 1, i.e., threads proceed at the same average speed. The second requires also Assumption 2, i.e., threads access shared data in the same order. This is common in regular loop code, for example, in matrix multiplication, even when a shared matrix is tiled. In addition, there may be multiple groups of shared data or different-distance thread-local reuses (for the same or different data). The CRI formula is then applied to each RI value.

3.4 Validation of Racetrack Assumption

The racetrack model assumes that the relative progress between different threads is uniform. We name such assumption *the relative thread progress uncertainty*. This has been proved in Theorem 3.2 using a Markov process model. In this section, we show an experimental validation of this assumption for one program, `gemm`.

The main idea of the experiment was to periodically send out a "stop-the-world" signal during a parallel execution for all threads to pause and dump the current progress. To record the progress, we modified the generated source code of `gemm` by adding a thread-local counter. This counter increments per inner loop iteration. We periodically send out a `SIGPROF` signal while the kernel is running. When the signal handler is triggered, it records the value of the thread-local iteration counter, and after all threads finish their logging, it resumes the thread execution. We repeat this experiment 10 times.

At each pause, we randomly selected a thread as the base point and calculated the differences between the progress counters. By first taking their differences and then computing their reduced modulo by the total iteration count of inner loops, we obtain the *relative thread progress*, measured by the number of iterations.

The experiments were conducted with 4 threads. To collect enough data, we set the data size to 1024 in `gemm`. Based on the problem size, we set the logging intervals of `gemm` to 100 microseconds. Since Theorem 3.4 is based on the Markov Chain model, it needs time to reach the stationary distribution. Therefore, we filtered out around 10% timestamps at the beginning. Additionally, we also removed the timestamps after exiting the parallel loop.

Figure 4 shows the progress of four threads and the distribution of relative thread progress in two example executions [4]. The thread progress is quite different. In the execution shown on the left, one thread lags behind the others. In the other execution, it runs ahead of the others. The relative thread progress, however, shows similar distributions in both runs, which are largely uniform. *This shows the uncertainty in relative thread progress.*

We also do the goodness-to-fit test using the chi-square test on the combination of these 10 runs against the uniform distribution. With p value being $0.23 (> 0.05)$, we have high confidence that our data conformed to the uniform distribution. Hence, the experimental result agrees with the theoretical conclusion (Theorem 3.2), and both have validated the assumption for the racetrack model (Theorem 3.4).

4 Implementation

To find RIs, we build on a recent technique called Static Parallel Sampling (SPS). [18] SPS analyzes sequential code and uses parallelization when analyzing multiple references (of the sequential code). To avoid confusion, we refer to the technique as either *static sampling* or SPS. Given a loop nest, SPS generates a *sampler* program. It has two parts. The first is *sampling a data access*. It randomly selects an iteration in the loop nest and records the memory address accessed by a reference. The second is *finding the reuse*. It includes a simplified loop code that "traces" memory access between the use and the reuse and counting the number of accesses. The tracing

[4] To save space, we select and present the two that show the most visible difference in thread progress.

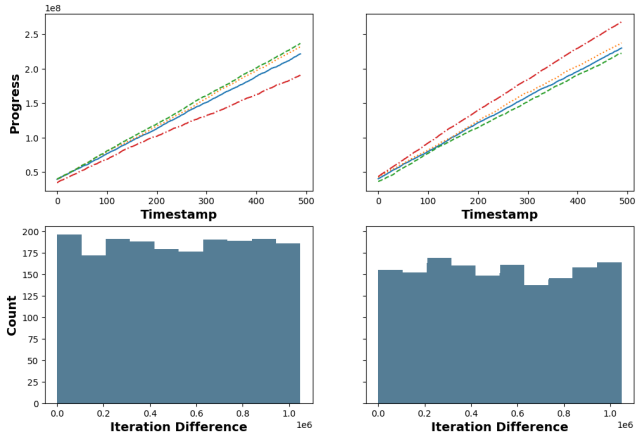

Figure 4: The progress of two four-thread executions (top two plots) and the distributions of relative thread progress (bottom). While thread progress differs between two runs, the distribution of their relative progress is similar (and largely uniform).

code iterates through a loop but does *not* execute it or create or access any data. It merely tracks data addresses referred in loop iterations.

We have developed PLUSS, which stands for *Parallel Locality analysis under Static Sampling*. From the PLUSS tool, we obtain symbolic concurrent reuse interval (*Symbolic CRI*) analysis. PLUSS adds the following new components to SPS to handle an OpenMP parallel loop:

Reference Classification. CRI model has two parts: The NB distribution for data that are private in each thread, and the Racetrack model for data shared among threads. One remaining problem is how to feed those PRI to the correct model. PLUSS addresses this issue via static analysis on array subscripts. In particular, only RIs generated by a reference with no parallel loop induction variable in its array subscripts will be passed to the Racetrack model. This classification is done before the sampler code generation and it can be extended to handle more general expressions of array indexing [3].

RI Sampler. Two more components are added in SPS sampler to support OpenMP loop parallelization: *Chunk Dispatcher* and *Thread Interleaver*. Figure 5 shows the sampler with these two modules. Initially, all threads are idle. The first module, *Chunk Dispatcher*, divides the parallel loop into fixed-size chunks and assigns these chunks to idle threads. Once assigned work, a thread is moved to the worker thread list. In the second module, Interleaver, worker threads generate the sequence of memory accesses, where reuse analysis collects RI histograms. When a thread finishes chunk, it returns to the idle thread list and is assigned new work. The sampler terminates when all threads are idle and Chunk Dispatcher has no more chunk.

The Sampler is used in two ways. For PLUSS, it collects thread-local RIs and feeds them to the model. For evaluation, it collects CRIs as a comparison with the model (in addition to CRIs collected by profiling).

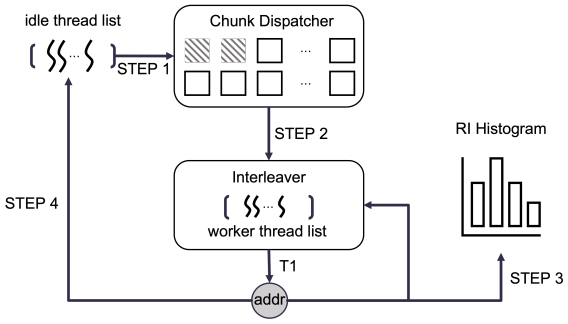

Figure 5: The workflow of the PLUSS sampler.

5 Evaluation

5.1 Experimental Setup

We have implemented PLUSS based on LLVM 11.1.0. It compiles an OpenMP program and generates a sampler program for each parallel loop, i.e. each loop nest that has a loop marked `#pragma pluss parallel`. A reference analysis pass is added to classify references before the sampler codegen pass. The negative binomial distribution and the racetrack model are implemented using the GNU Scientific Library (GSL) [28].

We evaluate PLUSS using PolyBench 4.2.1 [38], which has been extensively used in studying performance analysis [1, 45] and optimizations [2, 26]. We use 22 of the 30 programs. [5] We also test on two computation kernels, `dct` and `tce` from PLUTO [10], three image-processing applications, `anisotropic-diffusion`, `harris`, `edgedetect` and two machine-learning applications, `convolution` and `vgg` from [6, 43]. These benchmarks have 73 parallel loop nests, where 58 of them are in PolyBench and the rest in other sources. We use the `PPCG` compiler to parallelize these loop nests by inserting OpenMP directives [51]. We apply cache modeling after compiler parallelization and optimization. Optimization may change the locality and running time. When it affects all threads equally, Assumption 1 still holds, and the models remain valid.

For four-thread tests, we choose the SMALL_DATASET in PolyBench as the input size, which occupies around 128KB of memory. Experiments are conducted on a system equipped with an Intel Xeon Silver 4114 CPU @2.2GHz, which has 2 sockets, each with 10 physical cores. For 64-thread tests, we switch the input size to `LARGE_DATASET` to ensure all threads have chunks to run, and we collect the *PRI* on AWS `c6i.16xlarge` instances, which have 64 vCPUs (Intel Xeon 8375C (Ice Lake)) and 128GB RAM. Data arrays are double-precision (8 bytes). We target fully associative LRU caches with 64-byte cache blocks.

We compare PLUSS with a profiling tool implemented using the binary instrumentation tool, PIN [39]. It instruments every load and store, which, when executed, records the address and uses a critical section to maintain shared meta-data including a global time counter, a hashmap of last-access times, and the RI histogram.

[5] Among 30 benchmarks in PolyBench 4.2.1, 26 can be parallelized without data race. Four others, `cholesky`, `durbin`, `jacobi-1d` and `trisolv`, are excluded because they have short parallel loops (only the innermost loop is parallelized). We use the remaining 22 programs in our experiments.

When profiling a parallel program, most of the cost comes from accessing the shared meta-data, and, this cost increases with the thread count.

5.2 Accuracy

We compare PIN with the symbolic *CRI* model. We compare the miss ratio curves (MRC) generated by the proposed *CRI* model to the miss ratio curves from PIN profiling.[6] We also compare the *CRI* prediction with the MRC result by randomly interleaving memory traces of each thread. This interleaving scheme is adopted in PPT-SASMM [8]. Both the random thread interleaving scheme and the *CRI* model are implemented in the PLUSS compiler. In this evaluation, we refer to the first technique *PLUSS-RI* and the second *Symbolic-CRI* or *CRI*. Figure 6 compares the MRC of 29 test programs parallelized by four threads. To compare the exact and the approximate MRCs, we compute the *Mean Absolute Error (MAE)*: the average of absolute differences between two curves for all cache sizes considered, as used by others [15]. The following table shows the MAE of *CRI*.[7]

	PLUSS-RI	Symbolic-CRI	
	T=4	T=4	T=64
Accuracy[7]	97.18%	97.49%	93.16%

Basic CRI. As the accuracy table shows, *CRI* on average has slightly higher accuracy than the PLUSS-RI. In 16 out of 29 benchmarks, *CRI* and PLUSS-RI predict the same MRC because there is little inter-thread data sharing in these benchmarks. Under this circumstance, both *CRI* and PLUSS-RI assume threads will be chosen to execute in equal probability. Hence these two techniques would produce the same prediction. *CRI* outperforms PLUSS-RI in 10 benchmarks because it can better capture RI distributions with data intercepts. Data intercepts (see definition in Section 3.1) are more likely to happen in long RIs [56]. *CRI* employs static analysis to distinguish RIs formed by shared data and only long RIs will be passed to the Racetrack model. When interleaving accesses randomly, such information is lost and all RIs are assumed to be affected by data intercepts. Hence it predicts more short RIs, making the MRC cliff happens earlier. In three benchmarks, `correlation`, `covariance` and `tce`, *CRI* predicts worse than PLUSS-RI. This large gap is caused by the ignorance of data intercept caused by the spatial locality. For example, in `covariance`, `c0` is the induction variable for the parallel loop and `c2` for its inner loop. There are eight array elements inside one cache line. Reference `data[c2][c0]` lets the adjacent two threads access the same cacheline. Our *CRI* model assumes that data intercept rarely happens in spatial reuse because they are short. Therefore, all reuse pairs generated by this type of reference are passed to the NB distribution, leading to a higher miss ratio. We leave the spatial locality support in *CRI* for future work.

Symbolic CRI. Though the difference in accuracy between the MRC predicted by random interleaving and that by *CRI* is tiny, *CRI* is still better. *CRI* models data locality on shared cache with a symbolic thread number T. The thread number T determines the probability of success p in a negative binomial distribution, and the number of data intercepts n in the racetrack model. This feature enables *CRI* to predict the RI distribution of parallel programs with any thread counts without recollecting the *PRI*. We configure the *CRI* model to distribute the RI under $T = 64$ using the *PRI* collected under $T = 4$ and compare it with the miss ratio curves profiled by PIN (Figure 7). We switch the input size to `LARGE_DATASET` to ensure all threads have chunks to run, and we collect the *PRI* on AWS `c6i.16xlarge` instances, which have 64 vCPUs (Intel Xeon 8375C (Ice Lake)) and 128GB RAM. We still use the mean absolute error to quantify the accuracy. Symbolic *CRI* can achieve 93.16% mean accuracy. Compared with the 97.49% accuracy of *CRI* at $T = 4$, a 4.33% error is introduced by the symbolic analysis.

Two primary factors cause such accuracy degradation. First is the assumption of no data intercept in spatial reuse. The accuracy degradation caused by such ignorance has already been discussed in the previous paragraph, but it makes the symbolic analysis perform worse. The distance between the model-predicted RI and the observed RI for these spatial reuses positively correlates with the thread count, T. On the one hand, higher thread count T causes the NB distribution predicts longer RIs. On the other hand, as shown in Figure 3, higher thread counts are more likely to split the original reuse into smaller RIs. Such opposing influence on spatial reuse hurt the accuracy when applying the *CRI* model with a higher thread count. The second factor is the increased cold misses due to a higher thread count. For example, suppose our parallel task has 64 chunks, the 4 last chunks have no data reuse when running in 4 threads. When we increase the thread count to 64. Such cold misses increase 16x [8]. This change in cold misses will not be captured during the symbolic analysis since we do not recollect the *PRI* in the new thread count. Compared with the first factor, the mismatch in cold misses has less impact on the accuracy.

The results show that the two effects, dilation and intercepts, largely determine the parallel locality in all programs we have tested. While actual parallel executions are affected by numerous factors in software and hardware, just these two effects, when added to the analysis, produce a miss ratio curve that closely matches the profiling result.

5.3 PLUSS Speed

We measure the speed of two PLUSS techniques, PLUSS-RI and *CRI*, as the speedup over PIN. PLUSS-RI randomly interleaves memory accesses in each thread, so it is sequential. *CRI* is derived from *PRI*s, which can be done concurrently when collecting *PRI*s. For all versions, we use the average time of three runs. The speedup result is presented in Figure 8.

The geomean speedup by PLUSS-RI is 4.6x over PIN and *CRI* further improves the speedup to 5.8x. The speedup comes from multiple sources. First, PLUSS does not execute the target program: no floating-point computations or allocation of arrays. PIN, in comparison, has to compile and instrument the target program. Second,

[6] Note that, since instrumentation always introduces perturbation, there is no ground truth for the actual interleaving in a parallel execution.

[7] On average, MRCs have 37 points for cache sizes smaller than 64K, 16 points for cache sizes between 64K and 1M, and 20 points for cache sizes above 1M.

[8] Each thread has 2 chunks, and 64 chunks have no data reuse.

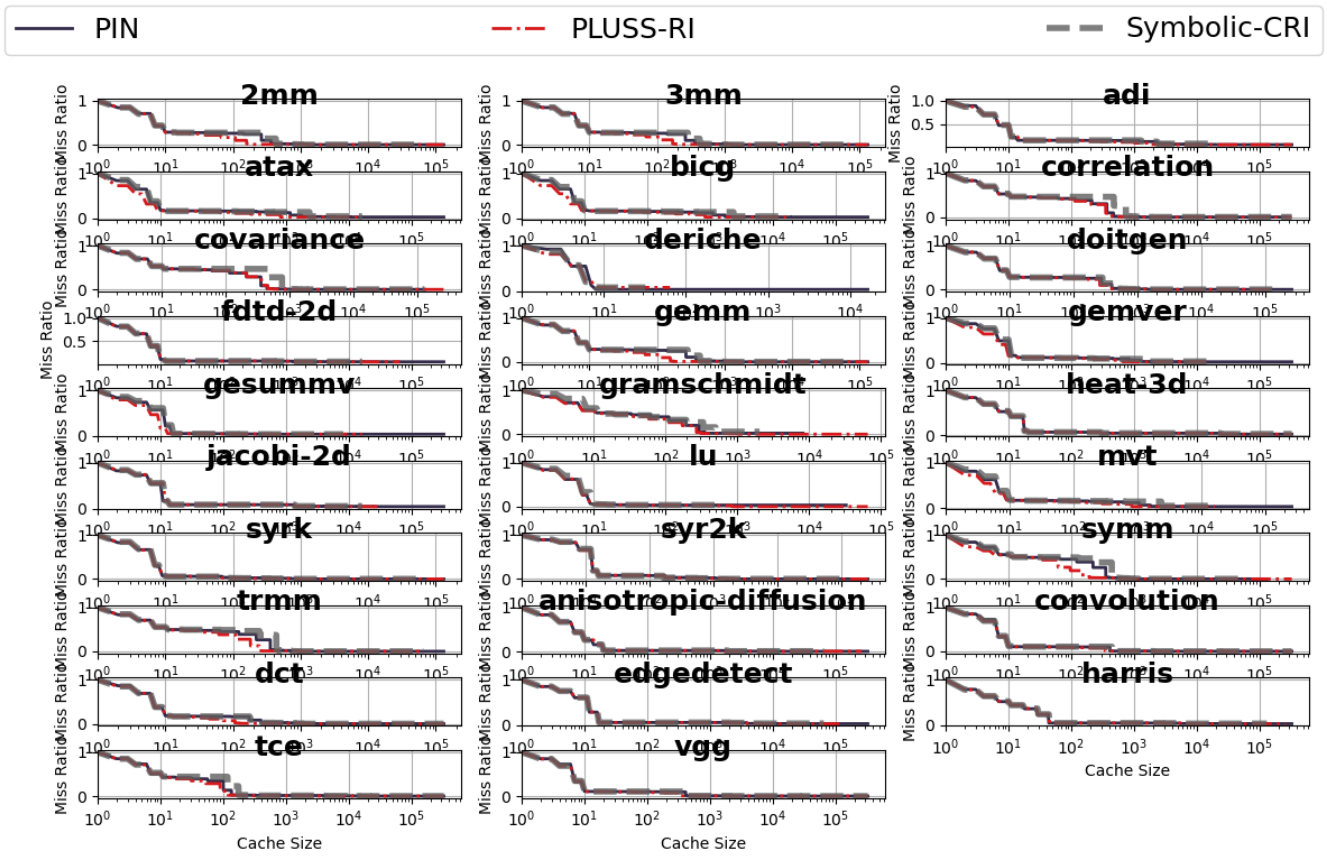

Figure 6: Miss ratio curves collected by the interleaving measured through PIN (black solid line), and predicted from random thread interleaving (red dashed line) and the *CRI* model (gray dash-dot line) for 29 programs. The cache size, c, is shown by the number of 64-byte cache blocks. All benchmarks run in 4 threads ($T = 4$).

PLUSS constructs or models a parallel execution, while PIN has to serialize the analysis of memory accesses using a global lock. There is the cost of locking in PIN (which increases with the thread count) but none in PLUSS. *CRI* embeds the statistical feature of thread interleaving in its model so that 1) there is no thread interleaving overhead and 2) collecting the *PRI* in each thread becomes an independent task and can be done in parallel. These two differences bring the additional 1.3x speedup in *CRI*.

Speedup by Symbolic Analysis. To predict the cache performance for a new thread count, PIN must rerun the program, and PLUSS-RI must recompile and rerun the sampler code. In contrast, *CRI* can *derive* the result by setting the new thread count (using the PRI collected for the previous thread count). Typically in our testing, *CRI* takes 0.14s for a new thread count, achieving 5805.6x speedup over PIN.

Furthermore, as PLUSS collects *PRI* in each thread, finding RIs in each worker thread can be conducted in parallel and it brings another 1.5x speedup in our tests.

5.4 Rustacean PLUSS

In this section, we use Rust both as a target for PLUSS analysis and as a tool for PLUSS implementation. Our model analyzes data parallelism in regular loop nests. In most of the paper, we analyze C/C++ code parallelized using OpenMP. Instead of parallelizing C/C++ using OpenMP, we parallelize Rust code using Rayon[9] as follows for `gemm`, where the `C` matrix is divided into chunks and run in parallel.

Listing 1: GEMM Kernel Code parallelized by Rayon.

```
C.par_chunks_mut(tmp / thread_num).enumerate()
    .for_each(|(tid, c_chunk)| {
    for i in 0..c_chunk.len() { // for each
        row in a chunk
        // GEMM operations
    }
});
```

[9]Rayon: A data parallelism library for Rust (Github Repository: https://github.com/rayon-rs/rayon, Version: 1.7.0)

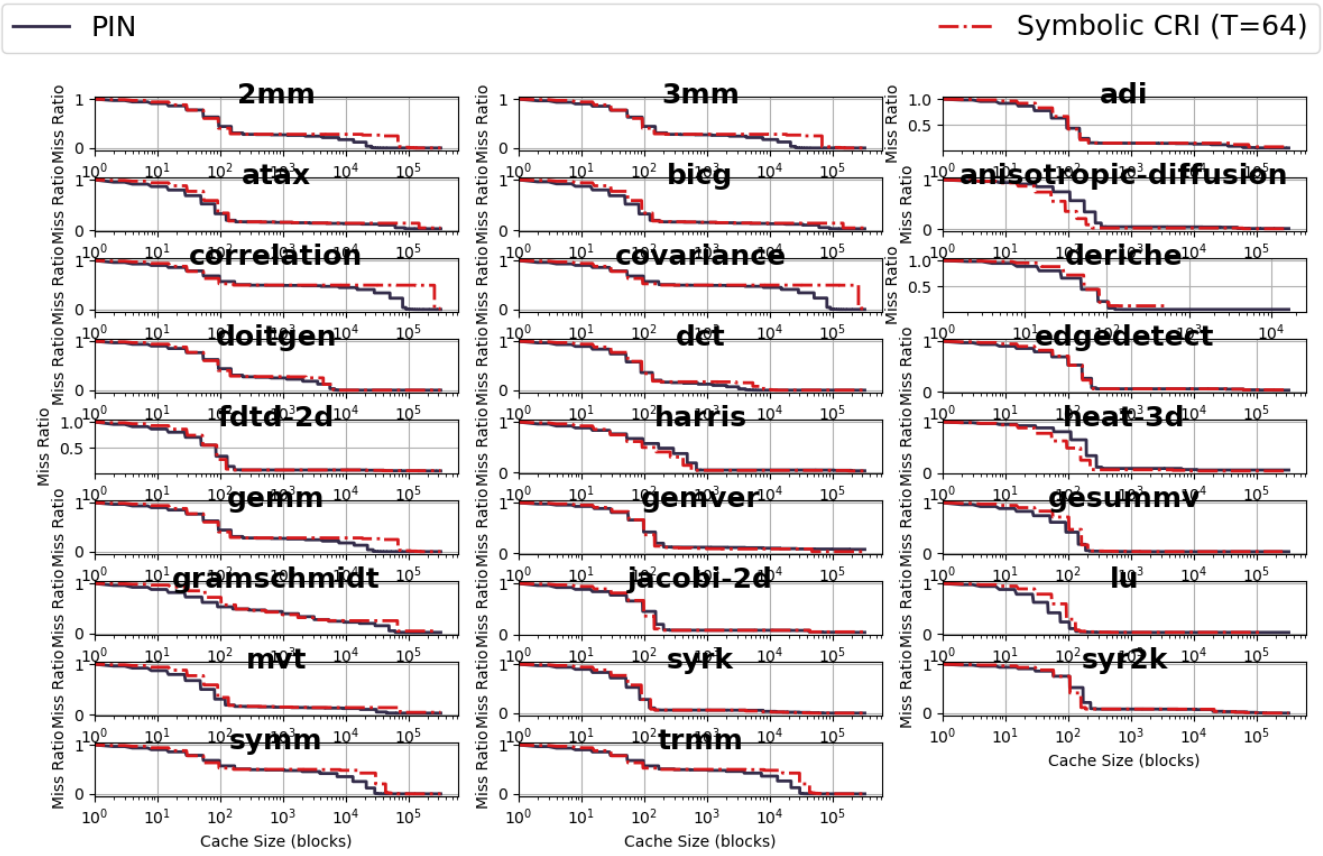

Figure 7: Miss ratio curves collected by the interleaving measured through PIN (black solid line), and predicted from the symbolic *CRI* model (red dashed line) for 26 programs. The cache size, c, is shown by the number of 64-byte cache blocks. All benchmarks are parallelized by 64 threads ($T = 64$). PIN reprofiles all benchmarks with the new thread count, but *CRI* predicts the miss ratio based on the *PRI* profiled with $T = 4$. Profiling `convolution` and `vgg` with 64 threads took too long (which we terminated after 20 hours), thus excluded from our results. `tce` is also excluded as it cannot be parallelized to fully use 64 threads when running with `LARGE_DATASET`.

PLUSS extracts PRI from each thread and analyzes data reuse at the source level.

C++ vs. Rust Based Samplers. The PLUSS compiler generates the sampler code in C++ currently. We have re-implemented by hand the sampler program in Rust for the test program `gemm` and studied the difference between Rust and C/C++ in correctness and performance.

The Rust compiler has more extensive and stringent checks on correctness. The following code examples show two places where the Rust version needs extra code that is not needed in C/C++. In the first example, a condition is added to tell the Rust compiler that there is no division-by-zero error. In the second, a branch case is added (to satisfy the compiler), although it is actually unreachable.

Listing 2: Rust code checking dispatcher boundaries.

```
if self.chunk_size != 0 {
  self.avail_chunk = if self.trip % self.
      chunk_size == 0 {
    self.trip / self.chunk_size
  } else {
    self.trip / self.chunk_size + 1
  };}
```

Listing 3: Rust dispatcher code.

```
let (cid, tid, pos) = if is_in_parallel_region {
  let cid = /* compute chunk id */;
  let tid = /* compute thread id */;
  let pos = /* compute position */;
  (cid, tid, pos)
} else {
  (0, 0, 0)};
```

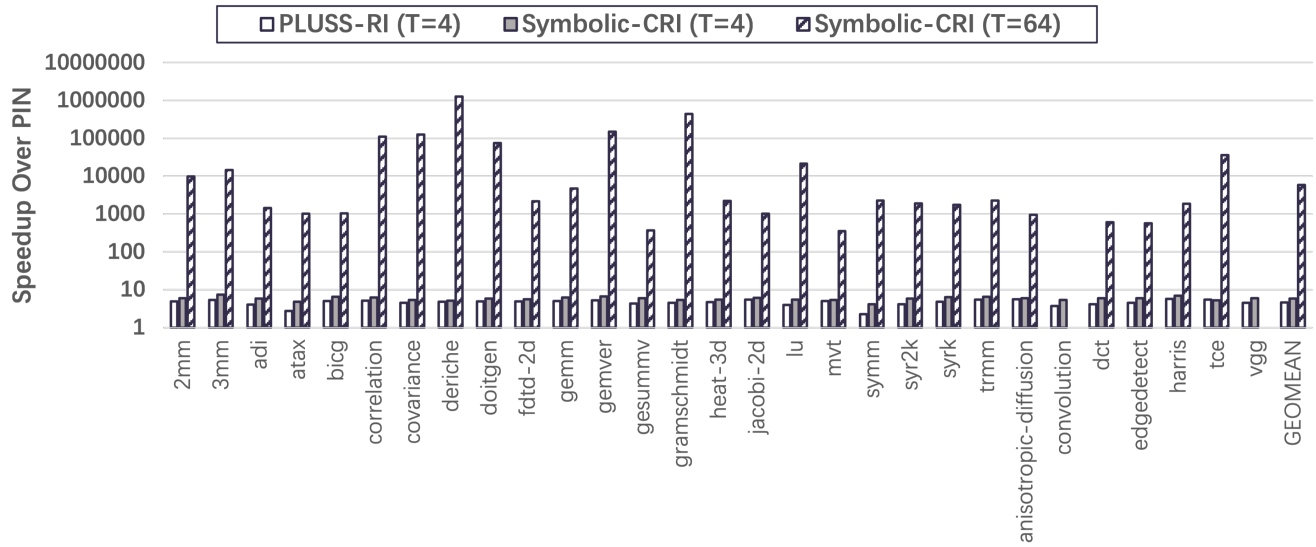

Figure 8: The speed of PLUSS-RI and two Symbolic *CRI* models, at $T = 4$ and $T = 64$, normalized to PIN. Profiling `convolution` and vgg with 64 threads took too long (which we terminated after 20 hours), thus excluded from our results. `tce` is also excluded as it cannot be parallelized to fully use 64 threads when running with `LARGE_DATASET`.

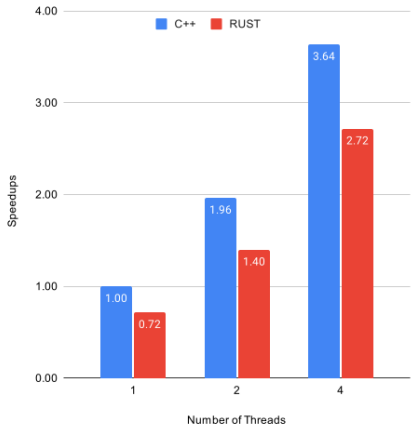

Figure 9: PLUSS in Rust is 30% slower than in C++ while having similar scalability

We have implemented the complete modeling and analysis in Rust and parallelized it using Rayon[10]. Fig. 9 compares the Rust modeler with the corresponding PLUSS code in C++, running with 1, 2, and 4 threads. The speed is normalized to single-thread PLUSS in C++. The Rust modeler is 30% slower, but the parallel speed increases at the same pace in both versions, i.e., they have similar speedups. The difference is not important in practice since the running times are short: 0.022 seconds for C++ and 0.029 for Rust, with 4 threads.

[10]We have also parallelized the Rust PLUSS using threads directly. The performance is not as good as Rayon, so we report Rayon results here.

Rust development helped to improve the C++ implementation in two ways. First, the C++ version now uses thread local storage (instead of using a global array), which further improves its speed and thread safety and eliminates a few cases of undefined behavior in the C++ code. In addition, we also implemented edge case handling and ensured all variables were initialized.

6 Related Work

Profiling. The cache performance of multi-thread code is traditionally analyzed by profiling, in particular, Concurrent Reuse Distance (CRD) [33, 48, 56] and *PPT-SASMM* [8] for CPU code and *PPT-GPU-Mem* for GPU kernels [5]. CRD predicts cache performance through reuse distance, which has $O(N \log M)$ complexity, where N is the number of memory accesses and M is the number of unique memory locations. PPT-GPU-Mem [5] instruments the binary to get the per-warp memory accesses, generates the shared trace with a Block-to-SM scheduling algorithm, and then builds the reuse profile for GPUs. PPT-SASMM [8] shares the same idea with PPT-GPU-Mem, but it generates the trace by instrumenting memory accesses in each basic block. Profiling of a parallel program often uses dynamic binary instrumentation tools such as PIN in this work as well as DynInst [13] and DynamoRIO [12]. To identify data reuses, a global lock is used and limits the scalability of profiling analysis. Cachegrind [44] simulates cache performance but uses no more than one CPU simultaneously.

Profiling is costly for parallel locality analysis because they require program instrumentation (except for RDX [52] and ReuseTracker [47]). For parallel code, their result is valid only for a single execution.

Shared Cache Locality. Unlike cache simulation, locality analysis shows the miss ratio of all cache sizes. For non-shared caches, earlier techniques used the reuse distance (the LRU stack distance) [57, 61].

For shared caches, composable analyses have been developed for program co-runs, where a set of sequential programs share the same cache [22, 57, 58]. These techniques are all based on reuse intervals. RI-based techniques are approximate in their prediction of the miss ratio. Many studies showed that the approximation is accurate [22, 54, 58, 59]. Chen et al. [17] cataloged several patterns when RIs may over- or under-predict the miss ratio. RIs are fast to measure and allow extremely efficient sampling [31, 52, 58]. However, these techniques assumed no data sharing between parallel tasks and hence cannot fully analyze multi-thread code.

Data Sharing. Two techniques measured the amount of shared data between threads in the entire run and then inferred its effect in smaller windows through probabilistic models [20, 33]. Analytical CRD characterizes the degree of sharing by constructing a Markov chain to compute the shared reuse profile for each thread [46]. While the Markov model assumes all threads share the same data, the shared footprint measures asymmetric data sharing, i.e. computing the amount of data shared by any thread subset from per-thread profiles [41]. However, it requires first profiling the program execution, and the parallelism is fixed and not symbolic.

Symbolic Analysis. A symbolic analysis can derive the locality for any thread count. For loop-based code, Wu and Yeung predicted the CRD and PRD profiles of any parallel execution by linear regression and extrapolation using profiled results from different thread counts and data input sizes [56]. Regression analysis shows the effect of parallel locality but is limited to linear relationships, unlike the patterns captured in this paper with negative binomial distribution for cache interference and the racetrack model for data sharing.

Parallel algorithm analysis shows the symbolic effect of processor count on computation and memory costs, for example, recently the effect of scheduling [49]. The memory cost is measured by the I/O operations based on the working set [23, 25, 30]. Our model is based on reuse intervals and captures specifically the (symbolic) statistical effect of thread interleaving and data sharing.

Compiler Analysis. For locality in sequential code, compiler analysis is based on dependence [3] or solving integer-set equations [7, 27, 29]. Compiler and profiling analysis have been combined to identify program-level causes of poor locality and do so across program inputs [9, 16, 24, 42]. To analyze parallel code, a compiler must statically model thread interleaving. For regular loops, PLUSS may use the compiler analysis instead of static sampling. PLUSS time complexity is linear, while reuse distance and integer-set equations are worse in asymptotic costs. In addition, it uses program information to reduce the need for tracing. For example, PLUSS is more efficient than PPT-SASMM [8] in its space cost. For the `2mm` benchmark with the same input size, PPT-SASMM reports 967MB memory usage (to store a trace) while PLUSS needs less than 1KB (to store the histogram).

7 Summary

This paper presents PLUSS, which performs locality analysis of OpenMP loops at compile time. Its two theoretical models, the negative binomial model of thread interleaving and the racetrack model of data sharing, largely capture parallel locality as observed from profiling. The analysis identifies locality effects at the program level and shows how they change with thread count. Finally, by being symbolic, it is one to two orders of magnitude faster than profiling. PLUSS analyzes data parallel code beyond those in OpenMP. The paper has demonstrated such a use in Rust and compares the parallel implementation in Rust and in C++.

Acknowledgments

We thank the anonymous reviewers of PACT'24 for their constructive feedback, Dong Chen, Sreepathi Pai for their help in the initial suggestion of the paper structure, Dylan McKellips, Arian Dokht Shahmirza, Leo Sciortino for their final proofreading, and Jionghao Han for her early participation in the Rustacean PLUSS implementation. This work was partially funded by the National Science Foundation (Contract No. SHF-2217395, CCF-2114319, CNS-1909099). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

References

[1] Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. 2021. PolyBench/Python: benchmarking Python environments with polyhedral optimizations. In *CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021*. 59–70.

[2] Aravind Acharya and Uday Bondhugula. 2015. PLUTO+: near-complete modeling of affine transformations for parallelism and locality. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. 54–64.

[3] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers.

[4] Pierre Amiranoff, Albert Cohen, and Paul Feautrier. 2006. Beyond Iteration Vectors: Instancewise Relational Abstract Domains. In *13th International Symposium on Static Analysis (SAS) (Lecture Notes in Computer Science, Vol. 4134)*, Kwangkeun Yi (Ed.). Springer, 161–180.

[5] Yehia Arafa, Abdel-Hameed A. Badawy, Gopinath Chennupati, Atanu Barai, Nandakishore Santhi, and Stephan J. Eidenbenz. 2020. Fast, accurate, and scalable memory modeling of GPGPUs using reuse profiles. In *ICS '20: 2020 International Conference on Supercomputing, Barcelona Spain, June, 2020*. 31:1–31:12.

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley (Eds.). IEEE, 193–205. https://doi.org/10.1109/CGO.2019.8661197

[7] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, and P. Sadayappan. 2018. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 32:1–32:26.

[8] Atanu Barai, Gopinath Chennupati, Nandakishore Santhi, Abdel-Hameed A. Badawy, Yehia Arafa, and Stephan J. Eidenbenz. 2020. PPT-SASMM: Scalable Analytical Shared Memory Model: Predicting the Performance of Multicore Caches from a Single-Threaded Execution Trace. In *MEMSYS 2020: The International Symposium on Memory Systems, Washington, DC, USA, September, 2020*. 341–351.

[9] Kristof Beyls and Erik H. D'Hollander. 2006. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of High Performance Computing and Communications. Springer. Lecture Notes in Computer Science*, Vol. 4208. 220–229.

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.

[11] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. 2021. A Survey of Cache Simulators. *ACM Comput. Surv.* 53, 1 (2021), 19:1–19:32.

[12] Derek L. Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph. D. Dissertation. USA. AAI0807735.

[13] Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.* 14, 4 (Nov. 2000), 317–329. https://doi.org/10.1177/109434200001400404

[14] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Trans.*

Archit. Code Optim. 11, 3 (2014), 28:1–28:25.

[15] Damiano Carra and Giovanni Neglia. 2020. Efficient Miss Ratio Curve Computation for Heterogeneous Content Popularity. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 741–751.

[16] Calin Cascaval and David A. Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the International Conference on Supercomputing*. 150–159.

[17] Dong Chen, Chen Ding, Fangzhou Liu, Benjamin Reber, Wesley Smith, and Pengcheng Li. 2021. Uniform lease vs. LRU cache: analysis and evaluation. In *ISMM '21: 2021 ACM SIGPLAN International Symposium on Memory Management, Virtual Event, Canada, 22 June 2021*, Zhenlin Wang and Tobias Wrigstad (Eds.). ACM, 15–27.

[18] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality analysis through static parallel sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 557–570. https://doi.org/10.1145/3192366.3192402

[19] Peter J. Denning. 1968. The working set model for program behaviour. *Commun. ACM* 11, 5 (1968), 323–333.

[20] Chen Ding and Trishul Chilimbi. 2009. *A Composable Model for Analyzing Locality of Multi-threaded Programs.* Technical Report MSR-TR-2009-107. Microsoft Research.

[21] David Eklov, David Black-Schaffer, and Erik Hagersten. 2011. Fast modeling of shared caches in multicore systems. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*. 147–157. *Best paper.*

[22] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 55–65.

[23] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2015. On Characterizing the Data Access Complexity of Programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 567–580. https://doi.org/10.1145/2676726.2677010

[24] Changpeng Fang, Steve Carr, Soner Önder, and Zhenlin Wang. 2005. Instruction Based Memory Distance Analysis and its Application. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 27–37.

[25] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-Oblivious Algorithms. In *Proceedings of the Symposium on Foundations of Computer Science*. 285–298.

[26] Stefan Ganser, Armin Grösslinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. 2017. Iterative Schedule Optimization for Parallelization in the Polyhedron Model. *ACM Trans. Archit. Code Optim.* 14, 3, Article 23 (Aug. 2017), 26 pages.

[27] S. Ghosh, M. Martonosi, and S. Malik. 1999. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems* 21, 4 (1999).

[28] Brian Gough. 2009. *GNU scientific library reference manual.* Network Theory Ltd.

[29] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A fast analytical model of fully associative caches. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 816–829. https://doi.org/10.1145/3314221.3314606

[30] Jia-Wei Hong and H. T. Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the ACM Conference on Theory of Computing*. Milwaukee, WI, 326–333.

[31] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. 2018. Fast Miss Ratio Curve Modeling for Storage Cache. *ACM Transactions on Storage* 14, 2 (2018), 12:1–12:34. https://doi.org/10.1145/3185751

[32] S. Jiang and X. Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. Marina Del Rey, California.

[33] 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 International Conference on Compiler Construction*. 264–282.

[34] Karl Rupp. 2018. 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

[35] David Levin, Yuval Peres, and Elizabeth Wilmer. 2017. *Markov Chains and Mixing Times: Second Edition.* American Mathematical Society.

[36] Pengcheng Li, Xiaoyu Hu, Dong Chen, Jacob Brock, Hao Luo, Eddy Z. Zhang, and Chen Ding. 2017. LD: Low-Overhead GPU Race Detection Without Access Monitoring. *ACM Transactions on Architecture and Code Optimization* 14, 1 (2017), 9:1–9:25. https://doi.org/10.1145/3046678

[37] Fangzhou Liu, Yifan Zhu, and Shaotong Sun. 2024. Parallel Loop Locality Analysis for Symbolic Thread Counts (Artifacts). https://doi.org/10.5281/zenodo.12738741

[38] Louis-Noel Pouchet and Tomofumi Yuki. 2018. PolyBench/C 4.2. http://https://sourceforge.net/projects/polybench/files/

[39] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 190–200.

[40] Hao Luo, Guoyang Chen, Fangzhou Liu, Pengcheng Li, Chen Ding, and Xipeng Shen. 2018. Footprint modeling of cache associativity and granularity. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 232–242. https://doi.org/10.1145/3240302.3240419

[41] Hao Luo, Pengcheng Li, and Chen Ding. 2017. Thread Data Sharing in Cache: Theory and Measurement. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 103–115. http://dl.acm.org/citation.cfm?id=3018759

[42] G. Marin and J. Mellor-Crummey. 2004. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 2–13.

[43] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*. 429–443.

[44] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 89–100.

[45] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2020. Automated derivation of parametric data movement lower bounds for affine programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). 808–822.

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

[47] Muhammad Aditya Sasongko, Milind Chabbi, Mandana Bagheri-Marzijarani, and Didem Unat. 2022. ReuseTracker: Fast Yet Accurate Multicore Reuse Distance Analyzer. *ACM Trans. Archit. Code Optim.* 19, 1 (2022), 3:1–3:25.

[48] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 53–64.

[49] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. Proactive Work Stealing for Futures. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 257–271.

[50] Kirshanthan Sundararajah and Milind Kulkarni. 2019. Composable, sound transformations of nested recursion and loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 902–917.

[51] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013), 54:1–54:23.

[52] Qingsen Wang, Xu Liu, and Milind Chabbi. 2019. Featherlight Reuse-Distance Measurement. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, 440–453. https://doi.org/10.1109/HPCA.2019.00056

[53] Wenwen Wang and Pei-Hung Lin. 2021. Does it matter?: OMPSanitizer: an impact analyzer of reported data races in OpenMP programs. In *ICS '21: 2021 International Conference on Supercomputing, Virtual Event, USA, June 14-17, 2021*, Huiyang Zhou, Jose Moreira, Frank Mueller, and Yoav Etsion (Eds.). ACM, 40–51. https://doi.org/10.1145/3447818.3460379

[54] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, Andrew Warfield, and Coho Data. 2014. Characterizing storage workloads with counter stacks. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. USENIX Association, 335–349.

[55] 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. https://doi.org/10.1145/2427631.2427632

[56] Meng-Ju Wu and Donald Yeung. 2011. Coherent Profiles: Enabling Efficient Reuse Distance Analysis of Multicore Scaling for Loop-based Parallel Programs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 264–275.

[57] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul M. Chilimbi. 2011. All-window profiling and composable models of cache sharing. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 91–102.

[58] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 343–356.

[59] Chencheng Ye, Chen Ding, Hao Luo, Jacob Brock, Dong Chen, and Hai Jin. 2017. Cache Exclusivity and Sharing: Theory and Optimization. *ACM Transactions on Architecture and Code Optimization* 14, 4, 34:1–34:26. https://doi.org/10.1145/

3134437
[60] Liang Yuan, Chen Ding, Wesley Smith, Peter J. Denning, and Yunquan Zhang. 2019. A Relational Theory of Locality. *ACM Transactions on Architecture and Code Optimization* 16, 3 (2019), 33:1–33:26.
[61] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems* 31, 6 (2009), 20:1–20:39.

A Artifact Appendix

A.1 Abstract

The paper presents a tool called Parallel Locality Using Static Sampling (PLUSS). PLUSS proposes new mathematical models to predict shared cache misses for parallelized regular loop nests, where the symbolic nature of the models makes it scale easily to massive thread counts. This appendix provides the necessary information for evaluating the artifacts of PLUSS. The submitted toolset consists of two parts:

(1) The main artifact of PLUSS, containing all the required files to produce the miss ratio curves using the baseline profiling method and our proposed models.
(2) The Rust extension package, as detailed in Section 5.4 of the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Profiling, Program Instrumentation, Symbolic Computation, Mathematical Modeling
- **Program:** Bash and Python 3.10 Scripts, Patched LLVM compilers, PIN profiler, Polybench.
- **Compilation:** C/C++ compilation toolchain, GNU Makefiles.
- **Transformations:** Not applicable.
- **Binary:** Not applicable.
- **Model:** Negative Binomial Distribution, Markov Chain, Racetrack Model.
- **Data set:** Polybench (4.2.1).
- **Run-time environment:** Bash, Python runtime.
- **Hardware:** Multicore processors (128 cores).
- **Run-time state:** Not database needed. Results are produced by logs.
- **Execution:** Managed by Bash and Python scripts.
- **Metrics:** Mean Absolute Error (MAE).
- **Output:** Reuse Interval Distribution and Miss ratio.
- **Experiments:** Profiling, sample and modeling on Polybench kernels.
- **How much disk space required (approximately)?:** 100GiB
- **How much time is needed to prepare workflow (approximately)?:** 8 hours
- **How much time is needed to complete experiments (approximately)?:** 72-96 hours
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Code inside the Zenodo repository is licensed under both the MIT License and the Apache 2.0 License. The original Polybench code is distributed under Ohio State University Software Distribution License.
- **Data licenses (if publicly available)?:** Sample data inside the Zenodo repository is licensed under both the MIT License and the Apache 2.0 License.
- **Workflow framework used?:** Not applicable.
- **Archived (provide DOI)?:** The data that support the findings of this study are openly available in Zenodo as `10.5281/zenodo.12632114`, reference number [37].

A.3 Description

A.3.1 How to access. The DOI to the Zenodo repository is 10.5281/zenodo.12738741. The associated polybench kernels are also distributed within the same repo.

A.3.2 Hardware dependencies. This evaluation demands powerful multi-core processors. The original experiments were conducted with

- dual sockets of Intel Xeon Silver 4114 CPU,
- 64 vCPUs of Intel Xeon 8375C on AWS c6i.16xlarge instances, and
- single socket of AMD EPYC 7773X CPU.

To reproduce the results, it is not necessary to have exactly the same device topology or CPU configuration. However, a relatively large number of cores are required to demonstrate the effectiveness of our model.

x86-64/AMD64 architecture is required.

A.3.3 Software dependencies. The following software with the recommended version number are listed below:

- PPCG compiler version 0.08.
- Rust compiler and Cargo (it is recommended to use rustup),
- GNU Scientific Library version 2.7.1.
- PIN profiler, version 3.17.
- Python package: `pandas`, `numpy`, `scipy`, `matplotlib`
- Microsoft Powerpoint or equivalent for plotting.

A.4 Installation

This part is for manual installation. All procedures can be automated via scripts. Please see notes.

For the PIN results:

- Download and install PIN Profiler. This can be done on Arch-Linux using AUR.
- Create CacheSim PIN Plugin:
  ```
  cd CacheSim
  make obj-intel64/memory_trace.so TARGET=intel64
  ```
  If you get `-Werror` problems. Modify the first line of `memory_trace.cpp` into:
  ```
  #pragma GCC diagnostic push
  #pragma GCC diagnostic ignored
  ↪ "-Wmaybe-uninitialized"
  #include "pin.H"
  #pragma GCC diagnostic pop
  ```
- Install ppcg and compile polybench:
  ```
  # It downloads from upstream but
  # one can use the files in the Zenodo repo
  ↪ instead
  bash make_ppcg.sh
  bash generate_poly.sh
  ```
- Install Rust and compile result merger:
  ```
  curl --proto '=https' --tlsv1.2 -sSf
  ↪ https://sh.rustup.rs | sh
  cd result-merge
  cargo build --release
  ```

For PLUSS results:

- **C++ PLUSS:**
  - Download and install the GNU Scientific Library.
  - Modify the following two variables in the given Makefile if installing the GSL to a custom path.
    ```
    # in Makefile
    GSL_INCLUDE_DIR=
    GSL_LIB_DIR=
    ```
- **Rust PLUSS:** It is expected that the Rust version of `gemm` PLUSS gives the same result as the C++ version while having slightly lower speed with the same number of threads.

For Rust extension:
No special setup is required other than installing Rust via https://rustup.rs/.

A.5 Experiment workflow

For the PIN results: **By default, only `2mm` is executed. Please see Section A.7.**

- Run Pin profiling
  ```
  # by default, it only runs the first program
  # then exit. Please remove the exit command
  # to run all experiments (change the plotting
  # script to figure6 or figure7 before executing
  # the bash-runner.sh
  bash bash-runner.sh
  ```
- Run result merger and analyzer
  ```
  bash merge.sh
  bash analyze.sh
  ```

The Miss Ratio Curve should be ready inside `out.pluss` directories. The result filename looks like `2mm-t4-pin-mrc.txt`.

For the PLUSS results: We provide a python script to compile and run all experiments in a single run.

- Run and generate all pluss experiments. The script needs to be run under the root of the repository.
  ```
  # create a bin/ folder
  mkdir bin/
  # run sampler code with 4 threads
  # and run each sampler 3 times
  #
  # this will generate data for Figure 6
  # and 8
  python3 scripts/pluss_run.py -t 4 -e 3
  # run sampler code with 64 threads
  # each sampler run 3 times
  #
  # this will generate data for Figure 7
  python3 scripts/pluss_run.py -t 64 -e 3
  # plot the MRC curves and compute
  # the MAE of each benchmark
  python3 scripts/figure6.py
  python3 scripts/figure7.py
  ```

The miss ratio curves of PLUSS sampler should be ready inside `pluss-result/`. The MAE metric will be displayed on the `stdout`. The execution time of each sampler is summarized in `pluss_polybench_{XXX}_sampler_time.csv` under the root of the repository, where XXX is the name of each benchmark.

We provide two python scripts to plot Figure 6 and Figure 7, but we do not provide any script to plot the execution time of pluss sampler and that of PIN profiler. Instead, we use Microsoft Powerpoint to generate Figure 8. To summarize the execution time data, one can run:

```
# -t option can take more than one value,
# e.g. -t 4 6 means gathing the timing
# info run by two different thread numbers.
python3 scripts/data_loader -t 4
```

This script will load the execution time of each benchmark and summarize them into a large csv file, `pin\_polybench\_time.csv` and `pluss\_polybench\_time.csv`, for PIN and PLUSS respectively.

A.6 Evaluation and expected results

For evaluation, the following items are to be considered:

- Time for collecting Profiling and Sampling.
- Accuracy of the predicted miss ratio curves.

Detailed data (time and accuracy) is provided in the paper. For the accuracy result, Mean Absolute Error (MAE), is used as the metric.

A.7 Experiment customization

A.7.1 Change Program Set.

PIN Profiler. PIN Profiler is very costly to run. By default, we only run a single `2mm`. Please go to `/pluss/scripts/pin_run/list.sh` and follow the comment there to enable all test cases.

PLUSS Profiler. To run PLUSS sampler from a subset of programs, simply add the program name after the `-p` option. For example,

```
# run 2mm PLUSS sampler only
python3 scripts/pluss_run.py -p 2mm -t 4 -e 3
```

A.7.2 Change Data Size.

PIN Profiler. We prepare two sets of data sizes in the artifact and they can be quickly set by `scripts/pin_run/size.sh`. To switch datasets, you need to manually comment out the other one in this script. By default, we set the data size to `SMALL DATASET`. When gathering the baseline MRC of $T = 64$, a switch to `LARGE DATASET` is required.

A.8 Notes

We have packaged the program to install all dependencies and generate all results (except the Rust extension). To do so:

```
docker pull archlinux:latest
docker run -it -v
↪ /path/to/pluss/source/directory:/pluss archlinux
↪ /bin/bash
cd /pluss
bash run-all.sh
```

Please see the Experiment Customization part to see how to adjust default configurations.