Locality Analysis through Static Parallel Sampling

Dong Chen
University of Rochester
Rochester, New York, United States
dchen39@cs.rochester.edu

Fangzhou Liu
University of Rochester
Rochester, New York, United States
fliu14@cs.rochester.edu

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

Sreepathi Pai
University of Rochester
Rochester, New York, United States
sree@cs.rochester.edu

Abstract

Locality analysis is important since accessing memory is much slower than computing. Compile-time locality analysis can provide detailed program-level feedback for compilers or runtime systems faster than trace-based locality analysis.

In this paper, we describe a new approach to locality analysis based on static parallel sampling. A compiler analyzes loop-based code and generates sampler code which is run to measure locality. Our approach can predict precise cache line granularity miss ratio curves for complex loops with non-linear array references and even branches. The precision and overhead of static sampling are evaluated using PolyBench and a bit-reversal loop. Our result shows that by randomly sampling 2% of loop iterations, a compiler can construct almost exact miss ratio curves as trace based analysis. Sampling 0.5% and 1% iterations can achieve good precision and efficiency with an average 0.6% to 1% the time of tracing respectively. Our analysis can also be parallelized. The analysis may assist program optimization techniques such as tiling, program co-location, cache hint selection and help to analyze write locality and parallel locality.

CCS Concepts • **Software and its engineering → Software performance**; **Source code generation**;

Keywords Static analysis, program specialization, locality

ACM Reference Format:
Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality Analysis through Static Parallel Sampling. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3192366.3192402

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192402

1 Introduction

The gap between processor and memory speeds has been growing since 1980 [8]. The recent Intel Core i7 processor has 4 ns compute latency but 100 to 200 ns memory latency [22]. The gap also exists on high-bandwidth accelerators such as GPUs. On NVIDIA GPUs, global memory access latency is around 600 ns on Fermi and around 300 ns on Kepler though compute operations complete in about 10 ns. This large gap between operation latency and memory access latency limits performance. One way to bridge the gap is to introduce a memory hierarchy and develop techniques to maximize reuse in faster memory. The other way overlaps computation with data movement. Both ways rely heavily on data access locality to maximize hit ratio in fast memory and minimize the bandwidth usage [18].

Locality is a program property that is useful to guide compiler optimization, runtime scheduling and hardware design. There are two ways to analyze data locality: dynamic tracing and static code analysis. Trace-based locality analysis operates on a memory access trace collected by running the program. With a memory access trace, locality metrics such as reuse distance [33], footprint [46] and average eviction time [26] can be obtained. Static code analysis analyzes reference indices and code structures to identify data access and reuse. Example techniques include uniformly generated set-based reuse analysis [44], reference group and loop cost functions [27], static stack histogram [10], cache miss equations [23], reuse distance equation [4], and static reuse distance (for MATLAB) [12].

This paper presents Static Parallel Sampling (SPS), a new technique for program locality analysis.

SPS is the first static analysis for working-set locality. Previous techniques are often based on reuse distances and require the measurement of reuse distance (LRU stack distance) for precise analysis [4, 10]. Expensive parametric counting [2, 41] is often leveraged to derive reuse distance. Recent work has shown that the working-set locality can be measured using reuse time [26, 46], which is the number

of accesses between two consecutive accesses to the same cache block. Reuse time is more efficient to measure than the reuse distance which counts the number of *distinct* accesses instead.

SPS is statistical. It only looks at a subset of memory accesses. This does not affect program correctness (unlike, say, dependence analysis which must examine *all* data accesses to be sound). Although sampling has been used for trace analysis, SPS is the first analysis to sample for locality at the program level rather than the trace level. By sampling, SPS can trade off accuracy for speed.

SPS can be parallelized. Sampling of different reuse times can be done in parallel even when the target program is inherently sequential.

Overall, SPS leverages the computational efficiency of reuse time based locality modeling and can generate miss ratio curves statically and efficiently. It reduces the complexity and extends the applicability of static analysis. The main contributions of the paper are as follows:

- We formulate the notion of compile-time enumerable expressions (Section 2). SPS can be applied to any program that is compile-time enumerable.
- We show how SPS uses the program structure in sampling, by ensuring both sampling efficiency (Section 3.1) and sampling correctness (Section 3.2).
- We show how SPS computes the reuse time in $O(1)$ time by combining compiler analysis with sampling analysis (Section 3.4).

The organization of the paper is as follows: Section 2 introduces the working-set locality and characterizes the code structure targeted by SPS. Section 3 describes the new techniques including static sampling for reuse times, symbolic reuse time derivation and different sampling methodologies. Section 4 compares SPS with existing work. Section 5 evaluates SPS on PolyBench and *bit reversal (fft)*, its overhead compared to tracing, and the speedup from parallel sampling. Section 6 discusses possible applicable scenarios and future directions.

2 Static Reuse Time Analysis

SPS measures the *reuse time* which is the number of *memory accesses between the use and the next reuse of a memory address.*

Recent locality research relates this reuse time to cache performance. In particular, using the histogram of reuse times, a set of studies compute locality metrics including a type of average working-set size called footprint [46], the averaged eviction time (AET) [26], and write locality [14]. For example, the footprint is composable and can derive shared cache miss ratios of a co-run program group from per-program footprints [26, 45]. Footprint can also derive performance for important cache organizations including associativity [31] and exclusivity [48].

All aforementioned techniques use trace-based analysis. The goal of SPS is to measure the reuse time histogram *statically* without trace analysis, which can then be used to compute locality metrics such as the footprint.

To compute the reuse time histogram from a trace, it is straightforward to scan the trace and keep recording the previous accessing time for each address. Static analysis, however, examines memory references to deduce memory accesses. The program structure, i.e. the surrounding loops, branches, and reference subscript expressions determine the memory accesses.

SPS analyzes the following class of programs:

- A program is a sequence of statements and loop nests. The loops may be imperfectly nested. A statement is treated as a degenerate loop with just one iteration.
- The program may have branches, i.e. structured `if`-statements.
- The expressions of loop bounds, strides, branch predicates, and array subscripts contain only loop index variables and constants.

If an expression satisfies the third requirement, we say it is *compile-time enumerable*. For example, if the variable i is a loop index with constant lower and upper bounds, then the expression $i + 1$ is compile-time enumerable. Compile-time enumerable is a more general property than compile-time constant. A constant expression is enumerable, but an enumerable expression may not be constant.

Compile-time enumerable is more general than the Static Control Part (SCoP) [3] defined in the polyhedral model. Expressions in a polyhedral model must be affine. An enumerable expression does not have to be affine. SPolly examined the applicability of the Polly tool for polyhedral compilation in the SPEC2000 benchmarks [19]. Of 1862 code regions (single-entry, single-exit with at least one loop) surveyed across nine benchmarks in SPEC2000, SCoP could analyze 275 regions (14.8%). In contrast, compile-time enumerable can handle regions that SCoP could not due to non-affine expression (1230 regions), non-affine loop bounds (840), non-canonical induction variables (384, loops that do not start at zero and are not incremented by one), overflow issues from unsigned comparisons (199), presence of function calls (532), and complex CFG (253, due to e.g. switch). But as in the polyhedral model, compile-time enumerable cannot deal with regions that contain aliasing (1093).

Listing 1 shows a bit reversal loop typically used in programs such as FFT. In this loop, for a constant n, variables i, j and $i2$ are all enumerable, the expression $i < j$ in the if-branch is also enumerable. Hence, references $a[i]$ and $a[j]$ are compile-time enumerable.

SPS assumes that all expressions determining memory accesses are compile-time enumerable. This assumption permits a compiler to generate the trace of all memory accesses. Note that enumeration is not the same as running a program.

Listing 1. Bit reversal loop

```
for (i=0, j=0, i2 = n >> 1; i<n−1; i++) {
    if (i < j) swap(a[i], a[j]);
    k = i2;
    while (k <= j) {
        j −= k;
        k >>= 1;
    }
    j += k;
}
```

A compiler can enumerate all memory accesses but does not have the values stored in the memory. Nor does the compiler perform any computation on these values.

Locality analysis for a compile-time enumerable program may be done by obtaining the trace of memory accesses and performing trace-based reuse time analysis. Trace-based analysis, however, does not utilize the program structure, i.e. the rich static information in control flow and memory reference. Tracing is sequential and requires having the whole trace. Instead, SPS combines program analysis and tracing to enable parallel sampling.

3 Static Sampling of Reuse Time

SPS samples the reuse time for each reference r_i by sampling its Iteration Space (IS). First, it samples a subset of iteration points v_i from the IS of r_i. Second, for each sampled v_i, the IS of all r_j to the same array which may form a reuse is searched to find the iteration point v_j containing the next reuse. If no such v_j exists for all r_j, there is no reuse period. However, if v_j exists, the reuse time is calculated from (v_i, v_j) and the histogram is updated.

Although the analysis can be done inside the compiler (in our case, LLVM [30]), our implementation uses a separate autogenerated program-specific binary for the analysis. Essentially, the LLVM compiler specializes the algorithm in Alg. 1 as described next in the Section 3.1 for the program being analyzed. We adopt this approach for two primary reasons: one, it decouples our implementation from a specific compiler, and two, placed in a separate process, the analysis is easier to parallelize. This autogenerated "sampler program" accepts a sampling rate as input and outputs the locality results (typically, the miss ratio) back to the program.

3.1 Specializing the Analysis

The algorithm for static sampling *SPSAnalyzer()* is shown in Alg. 1. At a high level, the algorithm samples the iteration space (without replacement) for a particular reference (r_i) and identifies all the other references ($r_j \in R$) that may access the same addresses (Lines 3–4). Recall that we must scan all these other references to find the first possible reuse of r_i.

If the compiler can deduce that all the other references are regular references, it can solve for the first reuse (lines 6–11). Essentially, the compiler specializes *rtCalc* (line 9) for each reference pair r_i, r_j by generating a symbolic formula which computes the reuse time given the actual iteration vectors v_i, v_j. The symbolic reuse time derivation is described in Section 3.4.

The value of v_j is obtained from $SolveReuse(r_i, r_j, v_i)$ for regular subscripts r_i and r_j. The regularity is identified by the SIV test [1] used in a modern compiler, which originally concentrates on deriving distance vectors [28, 29]. These array subscripts are separable affine expressions $aI+b$ where I is a loop induction variable, and a, b are constants. Each dimension of v_j can be derived by $(a_{r_i} I_{v_i} + b_{r_i} - b_{r_j})/a_{r_j}$ for element granularity. By combining cache line size and stride of I_{r_j}, the cache line reuse iteration can be derived [13]. In this case, only assignment statements need to be generated for v_j.

If the references are irregular, *SearchReuse* (Alg. 2) compares addresses to identify any reuse in iterations that execute later. In both cases, once the minimum reuse time ($rtMin$) has been determined, $rtHisto()$ accumulates the frequency of reuse time $rtMin$ in a hash table indexed by the reuse time (line 15).

Algorithm 1: SPS analyzer main function

```
1  Function SPSAnalyzer():
2      for SamplingCnt ∈ [0, MaxNUM) do
3          r_i, v_i = getSample();
4          R = {r|r may form reuse with r_i };
5          if all R references are regular then
6              rtMin = INF;
7              for r_j ∈ R do
8                  v_j = SolveReuse(r_i, r_j, v_i);
9                  rt = rtCalc(r_i,r_j,v_i,v_j);
10                 rtMin = min(rt, rtMin);
11             end
12         else
13             rtMin = SearchReuse(r_i, v_i, R);
14         end
15         rtHisto(rtMin);
16     end
17 End
```

Alg. 2 shows the algorithm for searching the next reuse for reference r_i starting from iteration v_i. It generates a temporal variable rt in line 2 and records the number of memory accesses that happen in each iteration in line 9. Lines 3–4 generate two loops which for every loop iteration following v_i checks every reference r_j until they find the next reuse. The reuse may be in a different loop. If the reuse does not exist, the reuse time is infinite.

Algorithm 2: Search for the next reuse

```
1  Function SearchReuse(r_i, v_i, R):
2      rt = 0;
3      for each iteration v_j following v_i do
4          for r_j ∈ R do
5              if addr(r_i, v_i) == addr(r_j, v_j) then
6                  return rt;
7              end
8          end
9          rt+ = AcsPerIter(v_j);
10     end
11     return rt;
12 End
```

3.2 Correctness

Static sampling is defined to be correct only when *each reuse time is sampled with equal probability*. As reuses are not uniformly distributed among references and loops, it is necessary to carefully choose the way to perform sampling in line 3 in Alg. 1.

Algorithm 3: Generating next sample

```
1  Function getSample():
2      if #IS is known for all references then
3          Random sampling;
4      else
5          Stride sampling;
6      end
7      return r_i, v_i;
8  End
```

Alg. 3 shows the code template for generating next sample *getSample*(). It uses either random sampling or stride sampling. We also call the latter uniform sampling.

If $\#IS$ is known for all references, random sampling is used, which enumerates each reference r_i until the number of samples is $\#IS_{r_i} * SRSR$, where $SRSR$ is the sampling rate. For each sample, a random iteration vector v_i is selected. In stride sampling, the stride is computed from $SRSR$ as $\frac{1}{SRSR}$. Then it samples all iterations that are this stride apart in the iteration space.

For the following cases, $\#IS$ is known for all references. First, if all loop bounds are constants. Second, if the bounds are affine expressions of the induction variables of the surrounding loops with specific forms such as triangular loops, $\#IS$ can be derived. In these cases, Alg. 3 uses random sampling. In fact, both random and stride sampling can be used, but Section 3.5 will show that random sampling is preferred.

Next, we prove the correctness of SPS analysis.

Correctness of Sampling For any reuse time rt with distribution P, the sampled distribution P' is statistically equivalent to P.

Informal Proof Alg. 3 samples iteration points in the iteration space with equal probability. As each sampled v_i can either contribute one reuse time or have no reuse. Sampling v_i is equivalent to sampling reuse time. The original distribution P and sampled distribution P' can be calculated by Equation 1:

$$P = \frac{\sum_{r \in R_{rt}} cnt_{r,rt}}{\sum_{r \in R} cnt_r}$$
$$P' = \frac{\sum_{r \in R_{rt}} cnt_{r,rt} \times \frac{\#S_r}{\#IS_r}}{\sum_{r \in R} cnt_r \times \frac{\#S_r}{\#IS_r}} \quad (1)$$

where R_{rt} is the set of all references that can produce reuses with time distance rt, R the set of all references, $cnt_{r,rt}$ the number of reuses with reuse time rt contributed by reference r, cnt_r the number of all reuses contributed by reference r, $\#S_r$ the number of samples taken for reference r, and $\#IS_r$ the size of the iterations space of reference r.

From Equation 1, the ratios $\frac{\#S_r}{\#IS_r}$ for all references r are the same, so they are canceled out, and hence the sampling is correct.

3.3 Analysis Optimizations

Several optimizations can be performed to improve the speed of the analysis.

Reference Grouping Algorithm 1 falls back to *SearchReuse* whenever irregular references are present. However, *SolveReuse* which derives v_j for r_j based on the sampled r_i and v_i (line 8 in Alg. 1) can be performed using only r_i and r_j and no other reference. Thus, all references need not be regular. Instead, we can partition references into two groups: regular and irregular. The regular references are explored using lines 8–10, while for the irregular set, *SearchReuse* is used.

Reference Filtering Line 7 in Alg. 1 checks all references to the same array which can be avoided if we leverage control flow information. Once a reuse time is found for r_j, all the references that happen after all accesses of r_j can be filtered out. This order can be extracted from the control flow. For any two references to the same array, if one is reachable from the other, but the reverse is not reachable. It means all accesses of a reference must happen before any accesses of the other reference. With this order information, reference filtering can be performed after finding one reuse time in line 9.

Regular Subscripts in Irregular References Irregular references may contain regular subscripts. By separating the regular subscripts from the irregular, we can use *SolveReuse* on the regular subscripts and *SearchReuse* on irregular subscripts to avoid enumerating regular references.

Bounded Search Since references are grouped into regular and irregular. The temporary minimal reuse time $rtMin$ in Alg. 1 obtained from the regular references can be passed to

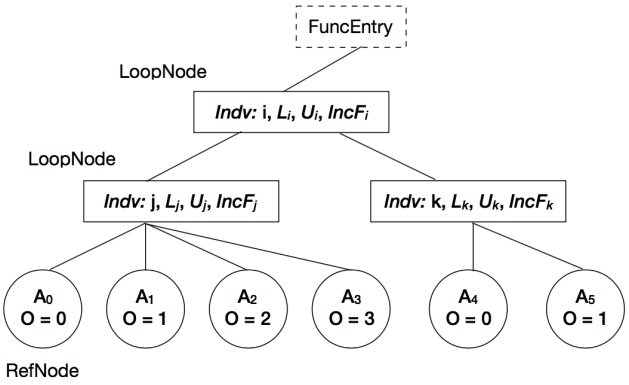

Figure 1. An example of the tree abstraction

SearchReuse() to terminate the search when rt in line 9 in Alg. 2 is larger than the temporary minimum.

Search Result Reuse In Alg. 2, samples from the same reference may produce the same reuse time. The samples following the first sample can use its search result to predict v_j without searching.

Parallelization For coarse-grained parallelism, each reference can be sampled independently. This may be too coarse-grained when the number of arrays are less than the number of hardware threads we have. For fine grained parallelism, all the samples can be processed independently by parallelizing the loop over different r_j in line 6 of Alg. 1 and line 4 in Alg. 2. The granularity can be adjusted according to the program characteristics and the hardware. Of course, $rtHisto(rtMin)$ must accumulate reuse time in the histogram correctly (e.g. by using atomics).

3.4 Symbolic Reuse Time Derivation

The generated SPS analyzer described in Section 3.1 solves or searches reuse iteration v_j of reference r_j for each sampled use iteration v_i of reference r_i. Given v_i, r_i and v_j, r_j, the naive solution to derive the reuse time from the iteration vectors is to traverse all iterations from v_i to v_j while counting the references in between. But this is inefficient as overlapping iterations may exist between different v_i, v_j for different (r_i, r_j) pairs or even different v_i, v_j for the same r_i. To avoid traversing and redundant counting for the same iterations, we introduce a program abstraction and a symbolic reuse time calculating expression construction algorithm for reference pair (r_i, r_j).

We abstract the program as a tree. Essentially, non-leaf nodes represent loops and leaf nodes represent references. The root is a dummy node representing the program entry. Pre-order traversal of the tree yields nodes in program order. A loop node contains loop info which is a list of induction variables, their symbolic bounds $[L_i, U_i]$ and their strides. A reference node stores reference information (array name and index expression) and its *reference order* O, which is the local order among references of the immediately enclosing loop. Two types of references do not matter when determining the reference order: references in different loop nests, and references in the same loop nest but with a deeper nesting. This tree is extracted after optimization before code generation of the target program.

Fig. 1 shows an example tree abstraction. Three loop nodes i, j, k where j, k are nested inside i. There are 6 reference nodes, where A_0 to A_3 are in the j loop with reference order from 0 to 3, A_4 and A_5 are in the k loop with reference order 0 and 1.

With this tree abstraction, for reference pair (r_i, r_j) where $r_i < r_j$ in pre-order traversal, the reuse time calculation expression can be derived by Alg. 4.

In initialization stage, each reference r can construct a path from the root *FuncEntry* node to leaf *RefNode* in form of [$LoopNode_1$, $LoopNode_2$, ..., $LoopNode_n$, $RefNode_i$], where each node in the list is a child node of the previous one. *Path*s are calculated for references r_i and r_j in line 2–3. Based on the *Path*s, *PrivatePath* which excludes the shared loop nodes between two *Path*s, *RoPP* the root of each private path and $lp_{RoPP_{r_i}, RoPP_{r_j}}/r_{RoPP_{r_i}, RoPP_{r_j}}$ the loop/reference nodes in the same level but between two roots of private paths are derived from line 4–10. *LCA* is the lowest common ancestor.

After initialization, the reuse time calculating expression *rtCal* can be derived by lines 12–33. For the loop nodes that shared by both $Path_{r_i}$ and $Path_{r_j}$, the iteration difference is determined by difference of the induction variables of the shared loops as line 12–14 shows. Then *InBetween* is derived in order to figure out the number of memory references occurred between $PrivatePath_{r_i}$ and $PrivatePath_{r_j}$ in one iteration of the last level shared loop (FuncEntry can be seen as loop with one iteration). Line 15–18 is to find the number of references happened in *PrivatePath* of r_i. Line 19–22 is to find the number of references happened in *PrivatePath* of r_j. Lines 23–28 find the number of the memory references between the end of $PrivatePath_{r_i}$ and start of $PrivatePath_{r_j}$. Lines 29–33 finalize the construction of reuse time calculating expression by adding or excluding the number of memory accesses in *InBetween* as *rtCal* in their shared loops always counts the number of memory references by one entire iteration. This will either over count or miscount the references in *InBetween*. For use at r_i, *InBetween* is miscounted because counting for the last iteration of shared loops stops at the point where r_i is in. As $r_i < r_j$ in pre-order traversal, the references from r_i to r_j in the last iteration is not counted. For reuse at r_i, *InBetween* is over counted for the same reason.

Complexity The formula generated for r_i, r_j contains $O(d)$ operations, where d is the deepest nesting for r_i, r_j. Assuming input programs have a constant bound on loop nesting, the formula has $O(1)$ time and space complexity.

Algorithm 4: Symbolic reuse time calculating expression construction

input : Tree abstraction, r_i, r_j
output: rtCal: Symbolic reuse time

1 ▷ *Initialization*;
2 $Path_{r_i} \leftarrow [lp_{i_1}, ..., lp_{i_n}, r_i]$; ▷ lp_{i_k} is the kth LoopNode for memory reference r_i
3 $Path_{r_j} \leftarrow [lp_{j_1}, ..., lp_{j_m}, r_j]$; ▷ lp_{j_k} is the kth LoopNode for memory reference r_j
4 $PrivatePath_{r_i} \leftarrow Path_{r_i} - Path_{r_i} \cap Path_{r_j}$;
5 $PrivatePath_{r_j} \leftarrow Path_{r_j} - Path_{r_i} \cap Path_{r_j}$;
6 $RoPP_{r_i} \leftarrow Root(PrivatePath_{r_i})$;
7 $RoPP_{r_j} \leftarrow Root(PrivatePath_{r_j})$;
8 $Peers \leftarrow children(LCA(r_i, r_j))$; ▷ *Peers* is a set of LoopNodes/RefNodes which are immediate children of the Lowest Common Ancestor of r_i, r_j
9 $lp_{RoPP_{r_i}, RoPP_{r_j}} \leftarrow \{lp | lp \in Peers \wedge RoPP_{r_i} < lp < RoPP_{r_j}\}$;
10 $r_{RoPP_{r_i}, RoPP_{r_j}} \leftarrow \{r | r \in Peers \wedge RoPP_{r_i} < r < RoPP_{r_i}\}$;
11 ▷ *Construction*;
12 **for** $lp \in Path_{r_i} \cap Path_{r_j}$ **do**
13 $rtCal$+=$(lp.indv_{reuse} - lp.indv_{use}) * AcsPerIter(lp)$; ▷ $AcsPerIter(lp)$ calculates the number of memory accesses for each iteration within LoopNode lp;
14 **end**
15 **for** $lp \in PrivatePath_{r_i}$ **do**
16 $InBetween$+=$(lp.U - lp.indv) * AcsPerIter(lp)$;
17 **end**
18 $InBetween$+=$AcsPerIter(lp_{r_{i_n}}.U) - O(r_i)$; ▷ $AcsPerIter(lp_{r_{i_n}}.U)$ is added when r_i does not share all the loops with r_j, AcsPerIter(lp) where lp in the same level but before r_i should be substracted;
19 **for** $lp \in PrivatePath_{r_j}$ **do**
20 $InBetween+ = (lp.indv - lp.L) * AcsPerIter(lp)$;
21 **end**
22 $InBetween$+=$O(r_j)$ ▷ AcsPerIter(lp) where lp in the same level but before r_j should be added;
23 **for** $lp \in lp_{RoPP_{r_i}, RoPP_{r_j}}$ **do**
24 $InBetween$+=$(lp.U - lp.L) * AcsPerIter(lp)$;
25 **end**
26 **for** $r \in r_{RoPP_{r_i}, RoPP_{r_j}}$ **do**
27 $InBetween$+=1;
28 **end**
29 **if** r_i *is use* **then**
30 $rtCal = rtCal + InBetween$;
31 **else**
32 $rtCal = rtCal - InBetween$;
33 **end**

An Example For example for the use at A_2 and reuse at A_4 in Fig. 1, the reuse time formula constructed by Alg. 4 is shown in Equation 2. $(i_{reuse} - i_{use}) * AcsPerIter(i)$ calculates iteration difference in the common loop in line 13. $AcsPerIter(i)$ can be derived by the loop stride and the bounds

```
void stencil (double* B, double* A) {
    for (int i = 1; i < 1025; i++) {
        for (int j = 1; j < 1025; j++) {
            B0 = A0 + A1 + A2 + A3 + A4;
        }
    }
    return;
}
```

$A_0 : A[i * 1026 + j]$	$A_1 : A[i * 1026 + j + 1]$
$A_2 : A[i * 1026 + j - 1]$	$A_3 : A[(i - 1) * 1026 + j]$
$A_4 : A[(i + 1) * 1026 + j]$	$B_0 : B[i * 1026 + j]$

Figure 2. 5-point stencil program

of loop j and k. With stride 1, $AcsPerIter(i)$ is $(U_j - L_j) * 4 + (U_k - L_k) * 2$. For *InBetween*, $(U_j - j_{use}) * 4 + 4 - O(A_2)$ is calculated in lines 13–18 and $(k_{reuse} - L_k) * 2 + O(A_4)$ is calculated in lines 19–22. No loops and references exist between loop j and loop k. Lines 23–28 are not executed. Give this reuse time formula, the SPS analyzer can calculate the reuse time in constant time given v_i, v_j.

$$\begin{aligned} rtCal =& (i_{reuse} - i_{use}) * AcsPerIter(i) \\ &+ (U_j - j_{use}) * 4 + 4 - O(A_2) \\ &+ (k_{reuse} - L_k) * 2 + O(A_4) \end{aligned} \tag{2}$$

With reuse time histogram, the miss ratio curve can be derived by reuse time based locality models [26, 46].

3.5 Uniform versus Random Sampling

Although the correctness of SPS is ensured statistically by using the same sampling rate, the sampling method can affect its accuracy. We evaluate two sampling schemes using the 5-point stencil program shown in Fig. 2.

Uniform sampling (U) with rates 0.098% and 0.99% (1024 and 10404 iteration points per reference) and random sampling (R) with rates 0.095% and 0.95% (1000 and 10000 iteration points per reference) are compared in Table 1. Trace based analysis result serves as a reference. rt is the reuse time, P and P' are the original and predicted distributions. The observations we have are: (1) for both uniform and random sampling, increasing sampling rate improves precision, (2) uniform sampling (unsurprisingly) fails to catch some of the reuse times, (3) compared to uniform sampling, random sampling yields better accuracy even with a lower sampling rate. Comparing R 0.095% with U 0.99%, random sampling even with 10x fewer samples can produce more accurate P' for most of the reuse times. For example, reuse time 6147 is introduced by reuse pair $(a[i * 1026 + j], a[(i - 1) * 1026 + j])$ when i equals to 1024. None of the samples in uniform sampling is from the last iteration of i as a constant stride is taken during the uniform sampling loop of i.

Table 1. Reuse time derived by tracing, static uniform sampling and static random sampling (element granularity)

	Trace analysis	SPS U 0.098%	SPS U 0.99%	SPS R 0.095%	SPS R 0.95%
rt	*P*(%)		*P'*(%)		
5	25	25.2	25.06	24.975	25
8	25	25.2	25.06	25	25
6135	24.98	24.4	24.82	24.975	24.99
6139	24.98	24.4	24.82	24.975	24.97
6140	0.02	0.8	0.24	0.075	0.015
6147	0.02	0	0	0	0.012

3.6 Number of Random Sampling Runs

Random sampling is more likely to produce better prediction than uniform sampling, but different randomly chosen samples for different runs may lead to different results.

To quantify the random effect, we vary the number of static sampling runs from 10 to 100 with random sampling rate 0.95%. The mean and standard derivation of distribution are measured from different runs. Then confidence intervals (CI) are calculated. The test is performed on 5-point stencil code shown in Fig. 2 for the cache line granularity which has more complicated reuse patterns than the element granularity has.

Fig. 3 shows the CI (95% confidence level) for each reuse time entry *rt* in the histogram. For the *rt* with distribution larger than 4.5% (1, 2, 4, 5, 6, 6117, 6121), the confidence interval ranges from around 0.12 pp (percentage point) to 0.31 pp. For the *rt* with distribution less than 0.02% (6127, 6129, 6133), the confidence interval is around 0.01 pp. Although encoding the cache line reuse makes the reuse time histogram more complex, random sampling by SPS can still construct the histogram precisely. As CI is stable when we increase the number of sampling runs, the following experiments are all done with only one static sampling run.

3.7 Per Array Analysis

The SPS framework presented so far targets all arrays in a program. It is also possible to adjust the framework to get reuse time histogram for arbitrary subset of arrays. This can be used to guide array placement and loop distribution or fusion. Only minor adjustments are needed to the static sampling algorithm: (1) *AcsPerIter* must be maintained on a per-array basis and the number of accesses to each array counted separately, (2) when deriving the symbolic reuse time calculating expression in Alg. 9, $AcsPerIter(lp)$ should be generated as the sum of the boolean variable for each array times its access count: $\sum_i (Bool_{array_i} * AcsPerIter_{array_i}(lp))$. Reference order can also be adjusted similarly. With the adjustments, reuse time can be calculated for any selected subset of arrays.

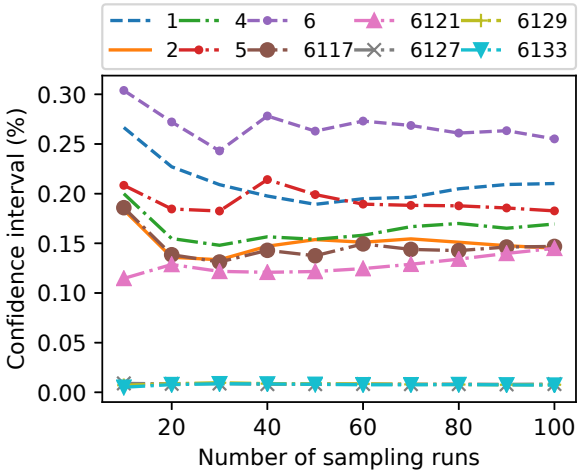

Figure 3. Confidence interval (Confidence level 95%) for each reuse time under different number of sampling runs with sampling rate 0.95%)

Fig. 4 shows the per array analysis result for 5-point stencil program in Fig. 2. Cache line granularity reuse time histogram (all the reuse times with distribution larger than 1%) for array A, B, A & B separately in subfigures (1), (2) and (3). By comparing (1), (2) and (3), we see that these two array references interfere with each other's reuses. For A, the reuse times that cross iterations are increased by 1 for reuse times 3, 4, 5 and 1020 for reuse times 5097, 5101. For B, all reuse time 1 are increased by 5. We see that the bars in the two-array histogram (3) are not simply repositioning the bars in single-array histograms (1) and (2). Subfigure (4) shows the miss ratio curves for array A and A & B (B is not shown since it drops to 0 with just 1 cache line). Compared to miss ratio curve for A & B, miss ratio curve for A drops faster at the beginning since the distribution for reuse times that are less equal to 5 for A is larger. When miss ratio curves drop to around 0.1 and become flat, miss ratios of A is slightly larger than those of A & B because the distribution for reuse times that are less equal to 6 for A is smaller. Finally, miss ratio curve for A drops to 0 earlier than that for A & B because the maximum reuse time for A is smaller than that of A & B.

4 Related Work

Locality analysis is difficult to perform statically due to the limited compile-time information. It's natural to limit the analysis to certain scopes by assuming certain code structure. Different assumptions and approaches exhibit different trade-offs among algorithm complexity, code structure coverage and applicability for different hardware (cache) architectures. Below, we first highlight five of the prior techniques and then summarize differences with SPS at the end.

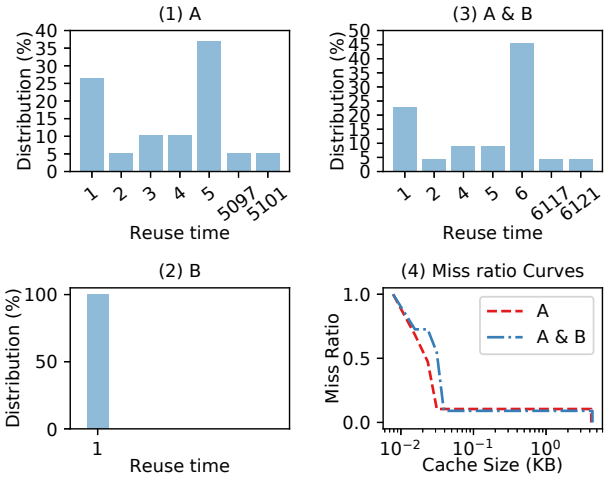

Figure 4. Per array reuse time histogram and miss ratio curve for 5-point stencil program: (1) reuse time histogram for array A (2) reuse time histogram for array B (3) reuse time histogram for array A, B (4) miss ratio curves

a. Loop cost functions [27] are formulated to predict the total number of cache lines accessed by loop nests. It provides a cost model which counting the cache line accessed in innermost loop and multiply the count by the number of iterations of all other outer loops. It classifies the memory references into groups before prediction to avoid over counting, as the same cache line may be accessed by different references in the same or different iterations of the innermost loop.

b. Uniformly generated sets based analysis [44] is formulated to measure the number of memory accesses per iteration for localized space L (i.e., inner L loops). The indexing functions for memory references are rewritten into the form of $H * \vec{i} + \vec{c}$, where $\vec{i}$ is the vector of induction variables, $\vec{c}$ is constant vector and H is the coefficient matrix. The uniformly generated set is defined as the group of memory references to the same array where H of the indexing functions are the same. Four types of reuses are modeled in order to count accesses precisely (the results are reuse vectors): Self temporal reuse R_{ST} for each reference is simply the kernel of H. Self spatial reuse R_{SS} for each reference is calculated by the kernel of H_S where H_S is H with all the elements in last row replaced by 0. For example if H is $\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$, H_S will be $\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}$ and the kernel is $span\{(0,1)\}$. Group temporal reuse R_{GT} is calculated for all reference pairs in the same uniformly generated set by the solution $\vec{r}$ that $H\vec{r} = \vec{c_1} - \vec{c_2}$. Similarly, group spatial reuse R_{GS} is determined by the solution $\vec{r}$ so that $H_S\vec{r} = \overrightarrow{c_{S,1}} - \overrightarrow{c_{S,2}}$. With all the reuses quantified, the number of memory accesses per iteration can be calculated by summing all the unique accesses.

c. Cache miss equations (CME) [23] are formulated to derive cache misses from reuse vectors [44] with consideration of cache size, cache line sizes, cache associativities and data size. The cold and replacement (i.e. conflict and capacity resp.) cache misses along reuse vectors are defined by linear equations. Two forms of code misses are formulated: (1) the present access is the first access along the temporal or spatial reuse vector, (2) two references along a spatial reuse vector access different memory lines. Replacement miss is formulated by assuming the memory lines are mapped to cache lines in a modulo fashion. CME requires counting the solutions of all equations. Sampling solutions of linear equations are adopted to reduce the overhead [39, 40].

d. Static reuse distance histogram [10] is formulated to derive reuse distance histogram for loop nests. A one pass analysis of the program yields the reuse distance histogram from which all cache size miss ratio curves can be generated. It uses the dependence distance to compute the LRU stack distance (reuse distance) histogram for nested loops. It first partitions iteration space in a way that all the incoming dependences for all iterations in that partition are the same. Then the volume of data accessed for iterations spanned by each dependence distance is calculated. With that, the minimal distance is accumulated to construct a histogram.

e. Reuse distance equation [4] is formulated to derive the reuse distance histogram for programs that can be represented by the polyhedral model. It is an integer set based framework containing two phases: reuse analysis which extracts the iteration points where the reuse occurs for a reference pair, and distance analysis which maps the iteration points to data space for all arrays to calculate the distance. The reuse analysis is performed by checking the index expression and loop bounds. Four conditions should be satisfied: (1) two references should happen within the iteration space, (2) reuse should happen after use, (3) reuse and use access the same memory location, and (4) no intervening accesses. The distance is calculated by another mapping from iteration space to data space based on the reuse set. For each element (iteration points for use and reuse) in the reuse set, the number of iterations in between is first derived. Then combining the data set that are touched for each iteration for each array, the distance is derived.

Comparison with Static Locality Analysis: For code structure coverage, a, b, c, and d above all target single loop nests with linear expressions and can handle symbolic loop bounds. However, d has stricter constraints as distance vectors are required for all references to obtain a precise reuse distance histogram. Given these constraints, SPS framework can use these techniques to handle symbolic bounds with *SolveReuse* and derive the symbolic reuse time histogram. Loop forests with constant bounds and branches with linear expression of the surrounding parameters are targeted by e. SPS targets compile-time enumerable codes which includes more code structures, such as non-linear subscripts.

For locality measurement, a, b both predict the total data usage measured by the number of data blocks. They can distinguish cases when optimizations can reduce the data usage to within the cache size. However, when the total usage is greater than the cache size, the usage itself does not predict the cache performance, i.e. the miss ratio. In contrast, c predicts misses for a specific cache size. It provides detailed cache performance but needs new prediction when cache size changes. To be machine independent, d, e predict reuse distance histograms. SPS predicts the reuse time histogram. Both types of histograms can derive the miss ratio curve.

In terms of complexity, a and b are in the order of the number of loops or references which is usually a small number while c, d, and e involve counting solutions of linear systems whose complexity may be polynomial [2, 41] or exponential [15, 17]. SPS is based on the reuse time and does not use solution counting. Using symbolic reuse time expressions, only O(1) time is needed to compute a reuse time. In the worst case, it searches all loop iterations, and the cost is linear to the length of the program execution.

Trace-based Locality Sampling: Given an execution, the reuse distance can be sampled by bursty sampling [49], parallelization and sampling [37], and hardware support [9, 38] Reuse distance is accurate in predicting the LRU cache miss ratio, but it is more costly to measure than the reuse time. SPS is based on reuse time. Many techniques measure the reuse time by sampling. They may be categorized by sample selection and mechanism. Samples may be selected by addresses [5], accesses [20, 26], or windows [25, 46]. The sampling may be done using compiler support [5] or binary instrumentation [25, 46]. Trace analysis precisely identifies reuses at block granularity, but it requires a program input to run, and its cost is proportional to the trace length. SPS also analyzes at block granularity. It is more efficient than tracing because (1) it enumerates data accesses and does not execute any computation or access array data, (2) it separates data at the program level and for example analyzes each array separately, (3) it uses the loop structure and can calculate the reuse time directly, and as a result (4) the analysis cost is less dependent on the input size as the tracing cost does. Next, we evaluate SPS and compare it with tracing.

5 Evaluation

Our implementation[1] of static sampling is based on LLVM 4.0.0. It first extracts loop and reference information to construct the tree representation of the program. Then it generates C++ static sampling code from the tree representation. The generation adopts search reuse described in Alg 2 optimized with search result reuse. Parallelization is done by generating C++ threads.

We evaluate our implementation on PolyBench [24] and *fft (bit reversal)* [6]. For PolyBench, linear algebra benchmarks (*3mm*, *gemm*, *atax*, *bicg*, *gemver*, *gesummv*, *mvt*, *syrk*) and stencil benchmarks (*convolution-2d*, *convolution-3d*) use input size 1024^2 for two dimensional arrays and 256^3 for three dimensional arrays. The *fft (bit reversal)* benchmark uses 2^{15} points as input. For *fft (bit reversal)*, the loop contains irregular control in branches in the loop body. Completely unrolling the loop control is needed before sampling.

From the sampled reuse time histogram, we compute the miss ratios using the footprint model, in particular the recent technique by Hu et al. (based on a "kinetic" model of cache and the average eviction time) [26]. It computes the same miss ratios as the higher order theory of locality (HOTL) [46]. These models in theory are not always accurate, but the accuracy has been evaluated experimentally for CPU caches (against hardware counter results) [46, 48] and storage caches [43]. In this evaluation, we focus only on the error of miss ratio curves introduced by the static sampling compared with the full trace profiling. To evaluate the precision, we compare the miss ratios converted from the sampling and from full-trace profiling. Both conversions use the same technique [26].

Only cache line granularity miss ratio curves are shown as cache line granularity reuses are harder to capture than element granularity. The cache line size is set to 64B, and the data element size is 8B in our test programs.

The precision and overhead of static sampling are measured for different Static Random Sampling Rates (*SRSR*s) and compared with trace-based analysis. Speedups of parallelized static sampling is measured by scaling the number of working hardware threads. All evaluations are done on Intel(R) Core(TM) i5-4260U CPU @ 1.40GHz with 4 GB memory and MacOS High Sierra (version 10.13).

5.1 Precision of Static Sampling

Five *SRSR*s range from 0.001 to 0.05 for each loop are tested, and the resulting cache line granularity miss ratio curves are shown in Fig. 5. As the loop iteration count is either 1024 or 256 for Polybench, the numbers of samples per loop are either 1, 2, 5, 10, 20, 51 or 0, 0, 1, 2, 5, 12. The number of samples per reference is the product of the number of iteration samples of its surround loops. For *fft (bit reversal)*, a single loop is used with the iteration count 2^{15}. The numbers of samples per reference are 32, 65, 163, 327, 655, 1638 for the five *SRSR*s. We compare to the cache-line granularity miss ratios measured by full-trace profiling.

At the lowest *SRSR* with one sample per loop for benchmarks in PolyBench and 32 samples for *fft (bit reversal)*, the sampled miss ratio curves mostly match the shape of the measured miss ratio curves. This shows the advantage of static sampling as it is guided by the code structure. The lowest-rate sampling shows similar sharp drops but not the precise points of the drop, and it can have large errors in the value of predicted miss ratios.

[1]The tool is available at https://github.com/dongchen-coder/symFP

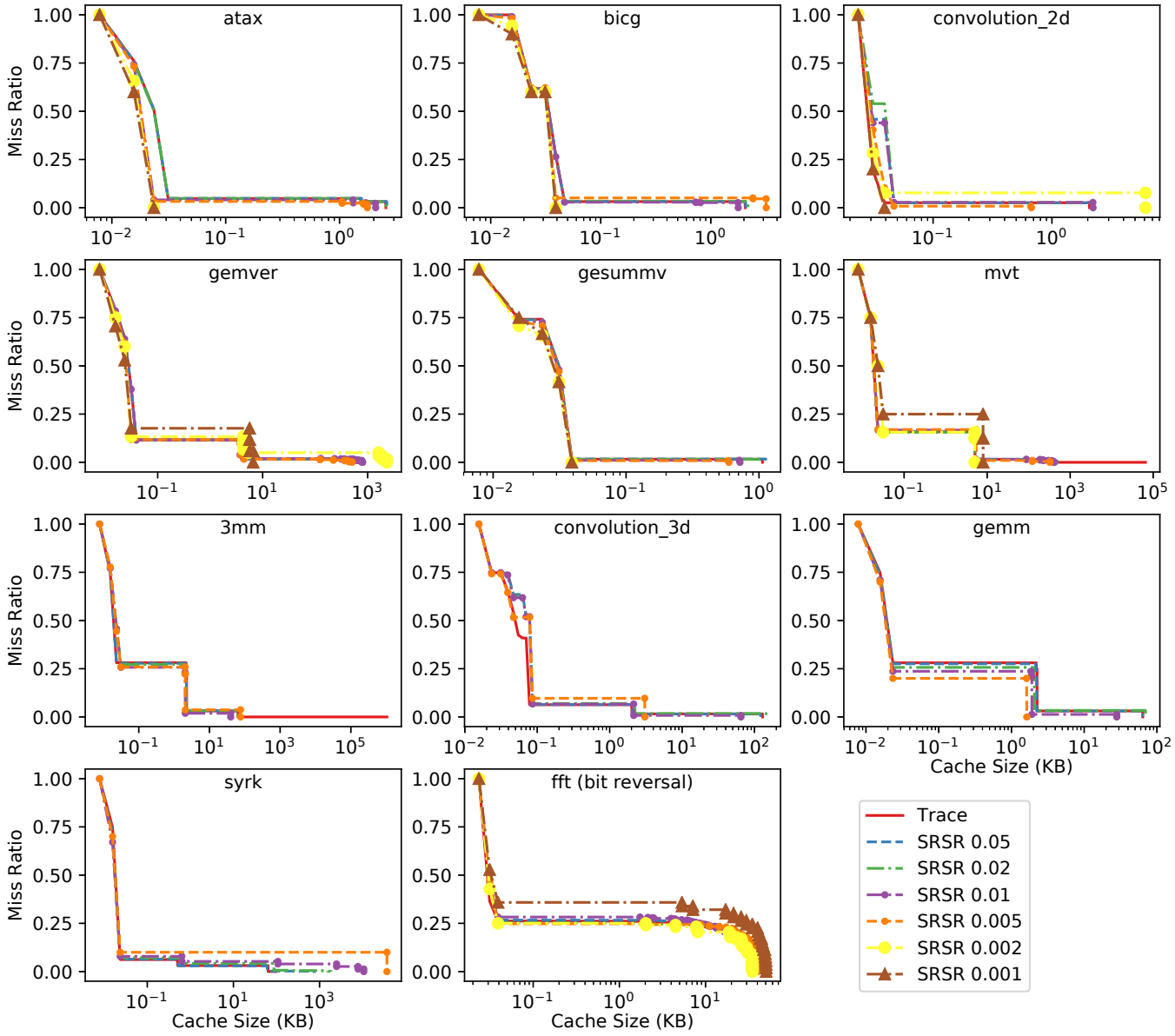

Figure 5. Predicted MRC by static sampling vs. by trace analysis

For 9 of 11 benchmarks (except *syrk* and *fft (bit reversal)*), the sampled miss ratio curve drops to zero earlier than the measured miss ratio curve due long reuse times. Since long reuse times account for a small portion of all reuses, they are difficult to capture at a low sampling rate, and sampling fails to detect misses at large cache sizes. In *syrk* and *fft (bit reversal)*, the sample curve drops to zero later than the measured curve. In these two programs, sampling successfully obtains the long reuse times but overestimates their proportion due to insufficient number of samples.

When the number of samples doubles from 1 to 2 per loop for benchmarks in PolyBench, sampling is mostly accurate with just a few exceptions. For example, in *syrk*, the measured curve drops to zero, but it is not until 82MB more cache later does the sampled miss ratio become zero. In *convolution_2d*, the sampled miss ratio is 5% higher than the actual for cache sizes between 128B to 1KB. As the number of samples increases, the predicted miss ratio curves become more and more precise. When *SRSR* reaches 0.02, the predicted curves are nearly exact matches for their respective measured curves.

In *gemver*, *mvt*, *syrk* and *fft (bit reversal)*, there are parts of the miss ratio curve that drop slowly (between sharp drops). A gradual drop indicates a large number of different

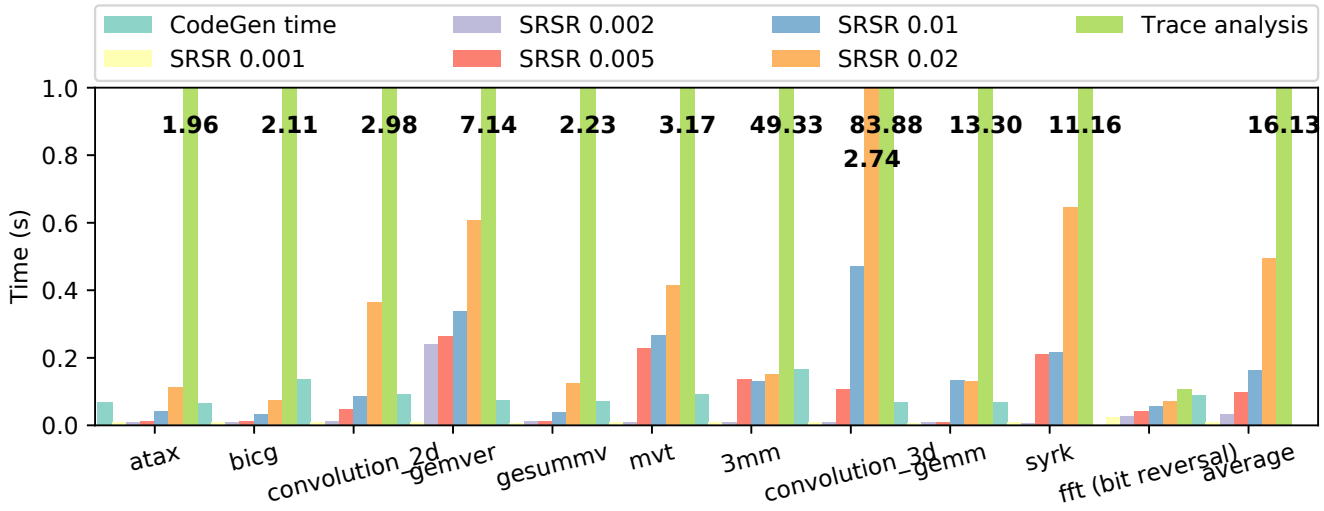

Figure 6. Overhead of code generation and static sampling vs. trace analysis

reuse times, which may happen even for one single reference. In such cases, it is not necessary to catch all different reuse times. The prediction is approximately accurate as long as we have sampled sufficient number of different reuse times. Whether the correct gradual drop will be discovered or whether the incorrect gradual drop will be corrected is based on the sampling of the reuse times. Low sampling rates cannot discover the gradual drop in *gemver, mvt, fft (bit reversal)* and cannot correct it in *syrk*.

5.2 Overhead of Static Sampling

Although static sampling does not run the target programs or require their inputs, its overhead may still be a problem if we want to apply it to time-consuming analysis such as auto-tuning or derive the miss ratio curve in online analysis.

The overhead of static sampling contains three parts: (1) code generation time which is the time taken by our LLVM-based tool to generate static sampling code from the source code of benchmarks, (2) compilation time which is the time taken by a standard C++ compiler to compile the static sampling code into the executable, (3) static sampling time which is the time of executing the sampler code to produce the miss ratio curve. As compilation time is usually small and depends on the compiler used, we do not to show it here. Fig. 6 shows the code generation (codeGen) time and static sampling time.

For code generation, the overhead is determined by the number of memory references that potentially introduce reuses and depth of their surrounding loops. Among all the benchmarks, *convolution_2d* and *convolution_3d* have the most references to check (10 and 16 references) which leads to larger code generation overhead. The maximum overhead for code generation is 0.167s which may be larger than static sampling time under small sampling rates. But generated SPS code is parameterized by the sampling rates and does not need to be regenerated when the sampling rate changes.

The static sampling time is measured with *SRSR* increasing by a factor of 20 from 0.001 to 0.02. The rate 0.05 is excluded because 0.02 can already produce nearly precise miss ratios. As a reference, the overhead of trace based analysis is also shown. Both static sampling and trace based analysis measure the miss ratios up to 20MB of cache. The execution time of static sampling is much lower compared to trace based analysis. On average, the time needed by static sampling is just 0.057%, 0.194%, 0.602%, 1.016% and 3.06% of the time of full trace analysis, when *SRSR* increases from 0.001 to 0.02.

The programs which have clusters of memory references to the same array in single loop nests are more likely to incur a higher overhead in static sampling, such as *convolution_2d*, *convolution_3d* with high *SRSR*. Because static sampling is performed for each reference, the number of samples for each loop is the sum of all samples of the enclosing references. Clustered references cause SPS to traverse the same loop iteration multiple times.

The programs which have long reuses, such as *syrk* and *gemver*, are more likely to incur higher overhead in static sampling since searching for reuses traverses more iterations.

5.3 Data Size Scaling

The number of samples for static sampling is determined by the complexity of reuse patterns which are a function of code structure and not the data size. This can further improve performance over tracing. To demonstrate, we scale the default data size by 2, 4 and 8 times. Overall, the results in Table 2 show that performance gain increases as the data size grows larger. In particular, we observe that the code containing more expensive arithmetic operations, such as multiplication in *3mm* and *gemm*, shows high speedups, since static

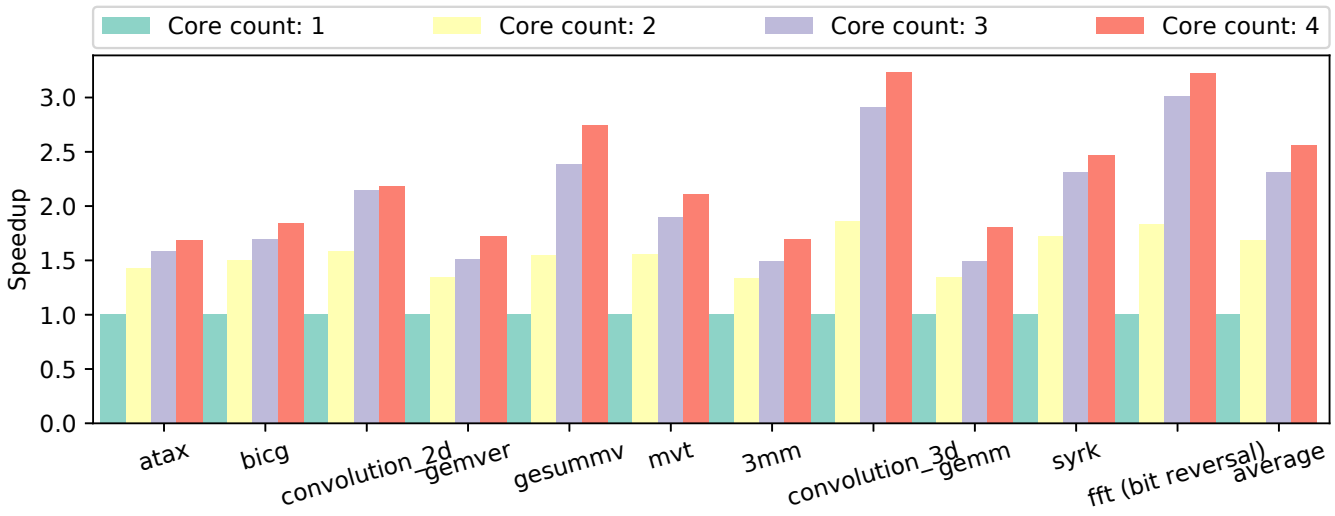

Figure 7. Speedups of static parallel sampling with the number of cores increasing from 1 to 4

Table 2. Speedups over tracing with data size scaling

benchmark	orig.	2X	4X	8X
atax	18.33	18.24	24.81	33.48
bicg	27.82	28.04	28.11	42.79
convolution_2d	8.63	8.29	9.13	9.19
gemver	17.30	13.65	20.09	34.12
gesummv	17.43	21.68	24.61	32.51
mvt	8.87	10.06	13.87	21.30
3mm	350	894	3132	10218
convolution_3d	33.14	31.05	45.13	71.21
gemm	95.03	164.64	316.96	756.34
syrk	18.06	48.34	122.64	343.53
fft (bit reversal)	1.47	1.86	1.82	1.82
geometric mean	20.97	26.83	40.15	66.20

sampling removes all these operations that are not related to the address calculation and program flow. Code that contains clustered memory references in a single loop nest has low speedups, such as *convolution_2d* and *convolution_3d*, since clustered memory references need more checks per loop iteration and may check the same iteration multiple times. Finally, code containing irregular control flow, such as *fft*, has low speedup since irregular control flow needs to be statically executed and cannot be accelerated with static information.

5.4 Parallel Sampling

The parallelization is performed by assigning one thread for each reference r_i in line 3 in Alg. 1. Fig. 7 shows the speedups of static sampling with *SRSR* 0.02 when increasing the number of threads. On average, parallel static sampling achieves 1.68, 2.31 and 2.55 speedups with 2, 3 and 4 hardware threads respectively. The amount of work per reference is irregular as it is determined by the number of samples, the number of surrounding loops, the search distance for each reuse, and the success rate of search result reuse. This irregularity leads to load imbalance limiting the speedup. Finer division of work by parallelizing SPS within a reference may improve scalability and make SPS more suitable for SIMD parallelism. We leave these improvements for future work.

6 Optimization and Future Work

SPS produces reuse time histograms and miss ratio curves precisely and efficiently during compilation, which can benefit compiler optimizations that use them. We next discuss this potential.

Tiling: Tiling is a compiler technique that improves the cache performance by reorganizing the iteration space into smaller chunks. The best tile size is usually selected based on the cache size, for example in Esseghir's tile size selection algorithm [21], Coleman and McKinley's TSS [16], Panda's selection algorithm [35]. With SPS, the tile size can be selected based on the miss ratio.

Co-run program cache sharing: Till now, trace based miss ratios curves [7, 42, 47] or hardware counter based miss rates [34] are used to guide program symbiosis [42] or cache partitioning [7, 34, 47]. These techniques improve cache sharing by reducing the interference among co-run programs. For example, the miss ratio curve can guide optimal cache partitioning for 4-program co-run groups on average to reduce total misses by 26% compared to free-for-all cache sharing and 98% compared to equal cache partitioning [7]. As SPS generates miss ratio curves at compile time with a low cost, it may improve these techniques by providing the programs' exact cache behavior before program scheduling or cache partitioning.

Generating cache hints: As SPS is a reference based analysis framework, it can provides reuse time histograms per reference. This can indicate the possible cache behavior for specific memory references. This gives us the opportunity to generate cache hints for each memory references efficiently. In [4], static cache hint selection achieves 10% speedup for a set of regular loops with Open64-compiler and Itanium processor.

SPS extracts reuse time during compile time. Reuse time is not limited to predict cache misses, it can also predict write backs [14]. Reuse time based model is composable [26, 46], it is possible to encode parallelism.

Measuring write locality: The emerging Non-Volatile Memory (NVM) technology combined with existing DRAM technology makes it possible to build larger capacity, low access latency and energy efficient memory systems by leveraging the strength of each technology. Hybrid memory systems are promising but need more sophisticated control of the data to reduce the number of long latency writes to NVM which can improve both NVM longevity and performance. Write locality [14] predicts the write backs for all cache sizes by combining reuse time with data access types (write/read). With these predicted write back curves, guided cache partitioning achieves 12%, 27%, 35% write back reduction for 2, 3, 4-program co-runs. By combining SPS with read/write information about data accesses, writebacks can be predicted by static sampling.

Locality in parallel code: Locality of parallel programs is hard to analyze due to non-determined interleaving of memory accesses introduced by parallel execution of the tasks without dependences. To bypass this hardness, declarative tuning [11, 36] for locality provides program scheduling constructs which is decoupled with the algorithm description. But when the scheduling space is large, parallel-locality model [32] is necessary to reduce the search space of scheduling. By combing SPS with parallel task dependences, it is possible to quantify parallel-locality.

Acknowledgments

The authors would like to thank John Criswell, Chucheow Lim, Chunling Hu, Liang Yuan, Kath Knobe, Zoran Budimlić, Hao Luo, Chencheng Ye, Xiameng Hu and Peng Zhao for the discussions. This research is supported in part by the National Science Foundation (Contract No. CCF-1717877, CCF-1629376, CNS-1319617), an IBM CAS Faculty Fellowship, and a fellowship from the Chinese Scholarship Council (No. 201403170421).

References

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

[2] Alexander I Barvinok. 1994. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research* 19, 4 (1994), 769–779.

[3] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. *CC* 6011 (2010), 283–303.

[4] Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223–250.

[5] 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.

[6] Paul Bourke. 1993. DFT (Discrete Fourier Transform) FFT (Fast Fourier Transform). *Internet, http://astronomy. swin. edu. au/~ pbourke/analysis/dft* (1993).

[7] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal cache partition-sharing. In *Parallel Processing (ICPP), 2015 44th International Conference on.* IEEE, 749–758.

[8] Carlos Carvalho. 2002. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation.*

[9] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. 2005. Multiple Page Size Modeling and Optimization. In *Proceedings of PACT*. 339–349.

[10] Calin Cascaval and David A. Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of ICS*. 150–159.

[11] Sanjay Chatterjee, Nick Vrvilo, Zoran Budimlic, Kathleen Knobe, and Vivek Sarkar. 2016. Declarative tuning for locality in parallel programs. In *Parallel Processing (ICPP), 2016 45th International Conference on.* IEEE, 452–457.

[12] Arun Chauhan and Chun-Yu Shei. 2010. Static reuse distances for locality-based optimizations in MATLAB. In *Proceedings of ICS*. 295–304.

[13] Dong Chen, Fangzhou Liu, Chen Ding, and Chucheow Lim. 2017. POSTER: Static Reuse Time Analysis Using Dependence Distance. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer.

[14] Dong Chen, Chencheng Ye, and Chen Ding. 2016. Write Locality and Optimization for Persistent Memory. In *Proceedings of the Second International Symposium on Memory Systems*. ACM, 77–87.

[15] Philippe Clauss. 2014. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 237–244.

[16] Stephanie Coleman and Kathryn S. McKinley. 1995. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of PLDI*. 279–290.

[17] George B Dantzig and B Curtis Eaves. 1973. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A* 14, 3 (1973), 288–297.

[18] C. Ding and K. Kennedy. 2004. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Parallel and Distrib. Comput.* 64, 1 (2004), 108–134.

[19] Johannes Doerfert, Clemens Hammacher, Kevin Streit, and Sebastian Hack. 2013. Spolly: speculative optimizations in the polyhedral model. *IMPACT 2013* (2013), 55.

[20] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *Proceedings of ISPASS*. 55–65.

[21] Karim Esseghir. 1993. *Improving data locality for caches.* Ph.D. Dissertation. Rice University.

[22] Naznin Fauzia. 2015. *Characterization of Data Locality Potential of CPU and GPU Applications through Dynamic Analysis.* Ph.D. Dissertation. The Ohio State University.

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

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

[25] Xiameng Hu, Xiaolin Wang, Yechen Li, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2017. Optimal Symbiosis and Fair Scheduling in Shared Cache. *IEEE TPDS* 28, 4 (2017), 1134–1148. https://doi.org/10.1109/TPDS.2016.2611572

[26] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 351–364.

[27] Ken Kennedy and Kathryn S McKinley. 1992. Optimizing for parallelism and data locality. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 151–162.

[28] David J Kuck, Yoichi Muraoka, and Shyh-Ching Chen. 1972. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Trans. Comput.* 100, 12 (1972), 1293–1310.

[29] Leslie Lamport. 1974. The parallel execution of DO loops. *Commun. ACM* 17, 2 (1974), 83–93.

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[31] Hao Luo, Guoyang Chen, Pengcheng Li, Chen Ding, and Xipeng Shen. 2016. Data-centric combinatorial optimization of parallel code. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 38.

[32] Hao Luo, Pengcheng Li, and Chen Ding. 2017. Thread data sharing in cache: Theory and measurement. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 103–115.

[33] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM System Journal* 9, 2 (1970), 78–117.

[34] Miquel Moreto, Francisco J Cazorla, Alex Ramirez, and Mateo Valero. 2008. MLP-aware dynamic cache partitioning. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 337–352.

[35] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D Dutt, and Alexandru Nicolau. 1999. Augmenting loop tiling with data alignment for improved cache performance. *IEEE transactions on computers* 48, 2 (1999), 142–149.

[36] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[37] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of PACT*. 53–64.

[38] David K. Tam, Reza Azimi, Livio Soares, and Michael Stumm. 2009. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of ASPLOS*. 121–132.

[39] Xavier Vera, Josep Llosa, Antonio Gonzalez, and Carlos Ciuraneta. 2000. A fast implementation of cache miss equations. In *Procs. of the 8th. Int. Workshop on Compilers for Parallel Computers*. 319–326.

[40] Xavier Vera and Jingling Xue. 2002. Let's study whole-program cache behaviour analytically. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*. IEEE, 175–186.

[41] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2004. Analytical computation of Ehrhart polynomials: Enabling more compiler analyses and optimizations. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 248–258.

[42] Xiaolin Wang, Yechen Li, Yingwei Luo, Xiameng Hu, Jacob Brock, Chen Ding, and Zhenlin Wang. 2015. Optimal footprint symbiosis in shared cache. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 412–422.

[43] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, Andrew Warfield, and Coho Data. 2014. Characterizing storage workloads with counter stacks. In *Proceedings of OSDI*. USENIX Association, 335–349.

[44] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of PLDI*. 30–44.

[45] 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 PPoPP*. 91–102.

[46] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *Proceedings of ASPLOS*. 343–356.

[47] Chencheng Ye, Jacob Brock, Chen Ding, and Hai Jin. 2017. Rochester elastic cache utility (recu): Unequal cache sharing is good economics. *International Journal of Parallel Programming* 45, 1 (2017), 30–44.

[48] Chencheng Ye, Chen Ding, Hao Luo, Jacob Brock, Dong Chen, and Hai Jin. 2017. Cache Exclusivity and Sharing: Theory and Optimization. *ACM Trans. on Arch. and Code Opt.* 14, 4, 34:1–34:26. https://doi.org/10.1145/3134437

[49] Yutao Zhong and Wentao Chang. 2008. Sampling-based program locality approximation. In *Proceedings of ISMM*. 91–100. https://doi.org/10.1145/1375634.1375648