Locality Analysis by Synthesizing Symbolic Reuse Intervals

Dong Chen
Independent Researcher

Abstract

As the complexity of the memory system grows, the demand to efficiently and precisely describe data movement is increasing. The past static techniques through complex mathematical models are often limited to linear expressions, and solving for a performance number requires approximation.

In this paper, we address the locality analysis problem with input-output-example-based program synthesis. We first propose three specifications to describe reuses and the structures of their input-output examples. Then for synthesizing, we adopt the unification search with a carefully designed elimination-free DSL, and reuse-type inferred, code-structure inferred biases. The produced programs summarize the locality. We report the synthesized symbolic reuse intervals for 30 programs in PolyBench. Predicted miss ratio curves compared with tracing are presented.

1 Introduction

High-Bandwidth Memory (HBM) and Non-Violate Memory (NVM) extend and enhance the traditional DRAM-only system with a larger size, lower energy consumption, non-violate, lower latency, and higher bandwidth. But efficiently and effectively using a variety of memory in a system is not an easy task. It is challenging for both system designers who want to make the complex memory system transparent to users and the application developers who want to manually achieve the hardware's peak performance. Intel Optane provides a software-defined memory to use NVM as part of the system memory, which is transparent to the operating system [17]. NVIDIA provides a unified memory interface for programming GPUs with HBM [1]. But at the same time, both of the vendors provide programming interfaces that allow some degree of program control from the programmers, like *clwb*, *clflush* instructions from Intel and *memory advises*, *prefetch* instructions from NVIDIA.

Though understanding and optimizing memory or cache performance is not a new task, its importance is growing when the complexity of the memory system increases. To characterize data access locality, there are three fundamental metrics: working set size (WSS) [12], reuse distance or LRU stack distance (RD) [24], and reuse interval (RI) [39].

Draft, 2022, Keep improving
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- WSS is the number of *distinct memory addresses* within a given length of a memory access trace.
- RD is the number of *distinct memory addresses* between two consecutive accesses to the same memory address.
- RI is the number of *memory accesses* between two consecutive accesses to the same memory address.

RI removed the requirement of distinction compared to RD. It enables collecting RI in linear time $O(n)$ while collecting *RD* needs $O(\mathtt{n}log(\mathtt{m}))$ time. n is the length of the memory access trace and m is the data size. Collected reuse interval/distance histograms (RIH/RDH) drives WSS. Their relation is summarized in [41]. With the above metrics, the performance of memory accesses is often presented in the form of miss ratios.

The other benefit of using RI instead of RD is that reuse intervals are composable. The interleaving of two memory access traces will only lengthen their RIs without changing the reuse relations if there is no sharing data between them. With composability, the analysis for RIs can be performed on each array in the program separately and merged afterward. This feature is handy for static analysis as we only need to focus on partial code related to a single array. In this paper, we focus on the static analysis based on RI.

Unlike static analyses that ensure the correctness of program transformations, static locality analysis can be statistical and does not require 100% precision to be valid. The problem can be formulated as discovering the relation between loop codes to locality metrics.

$$\mathtt{WSS/RIH/RDH} = f(\mathtt{loopCode})$$

All the past techniques [5, 9, 10, 13, 19, 38] have certain approximations for a limited set of loop structures. The state-of-art static approaches analyze loops that are Static Control Part (SCoP) [6] with the relaxation of allowing non-linear expressions. From the code, they build equations and solve for different loop-bound values. The solving process is usually expansive as they often require solving integer linear programming (ILP) problems [5, 8]. It can be seen as discovering the following relation:

$$\mathtt{WSS/RIH/RDH} = f_{\mathtt{ILP}}(\mathtt{loopCode}_{\mathtt{SCoP}})$$

Though the high-level intuitions are clear for static locality analysis, discovering a f that is accurate and solvable needs a lot of work. But with this abstraction, program synthesis may be an excellent candidate to discover f automatically with the proper encoding.

In this paper, we are addressing static locality analysis for loop-based code through the input-output-example-based program synthesis. We target `loopCodes` with branches but no indirect memory accesses and no data dependent controls. The high-level function f for locality we are trying to discover is shown as follows:

$$\texttt{Examples} \leftarrow \texttt{loopCode}$$
$$\texttt{RIH} = f_{\texttt{DSL}}(\texttt{Examples}).$$

Instead of building equations directly from the code, we first profile the code to generate a set of input-output examples to encode the relation between symbolic bounds $\vec{\texttt{B}}$ and reuse interval RI. Then from the `Examples`, we search for a program in a provided domain-specific language DSL that satisfies all input-output examples.

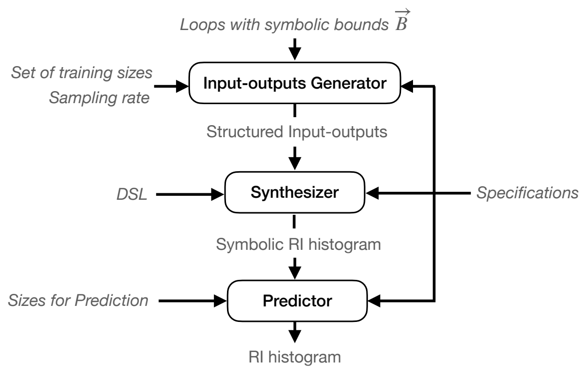


Figure 1. System framework

We design our system as Figure 1 shows. It contains three parts:

Input-output example generator. This generator takes the loop with symbolic bounds $\vec{\texttt{B}}$ and generates structured input-output examples for reuses. Different training sizes for $\vec{\texttt{B}}$ and different sampling rates for loop iterations can be provided.

Synthesizer. With the set of structured input-output examples, we run the synthesizer for each input-output example. The synthesizer takes a domain-specific language (DSL) and uses a bottom-up search with unification to find a program that satisfies all input-output examples. Different specifications will lead the synthesizer to find different programs. By merging all the synthesized programs, we get a symbolic reuse interval histogram for the loop with $\vec{\texttt{B}}$.

Predictor. With synthesized symbolic RI histogram, we can feed in different $\vec{\texttt{B}}$ values and directly calculate their RI distribution in constant time.

The organizations and contributions of this paper are listed as follows:

- We first exam the relation between code structure and reuse by dependence distance in Section 2. From the relation, we propose three specifications for $f_{\texttt{DSL}}$ and their input-output-example structures in Section 3.
- We then implement and present the first input-output-example-based program synthesis framework (tool) for static locality analysis in Section 4.
- Section 5 reports the symbolic reuse interval histograms for 30 programs in PolyBench. The overhead and precision are discussed.

2 Relation between data reuse and data dependence

Data Reuse. *Reuse happens when there are two consecutive accesses to the same memory location.* By scanning a memory access trace, we can easily identify all reuses. Figure 2a shows a 5-point stencil loop with five references to array a and one reference to array b. Figure 2c shows its memory access traces in element granularity (elm) and cache line granularity (cl). Eight elements share one cache line in cache line granularity. One reuse happens for element a[1][2] with reuse interval 5 (calculated by 7 - 2) in elm for the first eight accesses. In contrast, five reuses happen for cache line a[1][0~7] with reuse intervals 1 and 4 in cl. The RI distributions for the first eight accesses are 100% for 5 in elm and 75% for 1, 25% for 4 in cl.

Data dependence. *The data dependence exists between two references when references have accessed the same data with at least one write and there is a feasible run-time path from one to the other* [18]. Data dependence guides loop transformations and ensures the data is produced and consumed in the correct order. For the code shown in Figure 2a, data dependence only exists when different elements in array b share the same cache line.

It is useful to quantify the distance $\vec{v}$ between a dependence source reference $\texttt{Ref}_{\texttt{src}}$ at iteration $\vec{\texttt{I}}_{\texttt{src}}$ and its sink reference $\texttt{Ref}_{\texttt{snk}}$ at iteration $\vec{\texttt{I}}_{\texttt{snk}}$ by the number of iterations in between.

$$\vec{v}^{\,\texttt{Ref}_{\texttt{snk}}}_{\,\texttt{Ref}_{\texttt{src}}} = \vec{\texttt{I}}_{\texttt{snk}} - \vec{\texttt{I}}_{\texttt{src}}$$

Figure 2b shows the dependence distances for all references that read array a. They will not form a dependence as there is no write. But it is useful when we are trying to connect reuses with dependences.

Relation. Data reuse and data dependence share two requirements: (1) Both of them require at least two memory accesses. Reuse requires exact two accesses, and dependence requires at least one access for each statement (reference). (2) The location of the two memory accesses should be the same.

Besides shared requirements, both concepts have conditions that distinguish one from the other. Reuse requires consecutive accesses, while dependence does not care about

(a) 5-point stencil kernel

```
// a, b are matrices with size (B+2)*(B+2)
for (int i = 1; i < B+1; i++)
    for (int j = 1; j < B+1; j++)
        b[i][j] = a[i][j]+a[i][j+1]+a[i][j-1]
                 +a[i-1][j]+a[i+1][j];
```

(b) Dependence distance (read-after-read dependence [27])

Snk/Src	a[i][j]	a[i][j+1]	a[i][j-1]	a[i+1][j]	a[i-1][j]
a[i][j]	-	(0, 1)	-	(1, 0)	-
a[i][j+1]	-	-	-	(1, -1)	-
a[i][j-1]	(0, 1)	(0, 2)	-	(1, 1)	-
a[i+1][j]	-	-	-	-	-
a[i-1][j]	(1, 0)	(1, 1)	(1, -1)	(2, 0)	-

(c) Access traces for element-granularity (elm) and cacheline-granularity (cl)

i:	1	1	1	1	1	1	1	1	...
j:	1	1	1	1	1	1	2	2	...
References:	a[i][j]	a[i][j+1]	a[i][j-1]	a[i-1][j]	a[i+1][j]	b[i][j]	a[i][j]	a[i][j+1]	...
Logic time:	1	2	3	4	5	6	7	8	...
Trace (elm):	a[1][1]	a[1][2]	a[1][0]	a[0][1]	a[2][1]	b[1][1]	a[1][2]	a[1][3]	...
Trace (cl):	a[1][0~7]	a[1][0~7]	a[1][0~7]	a[0][0~7]	a[2][0~7]	b[1][0~7]	a[1][0~7]	a[1][0~7]	...

Figure 2. Relation between data reuse and data dependence

intervening accesses from other references. Dependence requires at least one write, while reuse does not care about whether the access is a read or a write.

By comparing those two concepts, we summarize their relation: *reuse* happens at the dependence sink with the shortest dependence distance, and *dependence* happens when there is at least one write in a cross-reference reuse.

As Figure 2b shows, we analyze the dependence distance $\vec{v}$ for all references to array a (read-after-read dependence [27]). For the column a[i][j+1] as source reference, there are three sink references with positive distance vectors: (0,1) for a[i][j], (0,2) for a[i][j-1], and (1,1) for a[i-1][j]. The shortest dependence distance is (0, 1) so the reuse is formed between a[i][j] and a[i][j+1] for elm.

Based on the dependence distances, we can easily define *shortest dependence distance* $\overrightarrow{sv}_{\mathsf{Ref}_{\mathsf{src}}}$.

$$\overrightarrow{sv}_{\mathsf{Ref}_{\mathsf{src}}} = \min_{\forall \mathsf{Ref}_{\mathsf{snk}}} (\vec{v}\,^{\mathsf{Ref}_{\mathsf{snk}}}_{\mathsf{Ref}_{\mathsf{src}}})$$

It is the minimal distance among distances $\vec{v}$ from the same source reference $\mathsf{Ref}_{\mathsf{src}}$ to all its sink references $\mathsf{Ref}_{\mathsf{snk}}$s. It indicates element granularity reuses.

A careful reader may notice that for different iterations, the number of valid $\vec{v}\,^{\mathsf{Ref}_{\mathsf{snk}}}_{\mathsf{Ref}_{\mathsf{src}}}$ will be different. It means that for different iterations, $\overrightarrow{sv}$ will be different even for the same source reference $\mathsf{Ref}_{\mathsf{src}}$. For example, for $\mathsf{Ref}_{\mathsf{src}}$ a[i][j] in Figure 2b, there are two sink references a[i][j-1] and a[i-1][j]. For iteration $1 \le i \le B, 1 \le j < B$, $\overrightarrow{sv}$ equals to (0,1); for iteration $1 \le i \le B, j = B$, $\overrightarrow{sv}$ equals to (1,0) as (0,1) is not valid for these iterations.

Unlike element-granularity reuse, cache-line-granularity reuse can happen between different array elements if they share the same cache line. It can be formulation by extending the above equation with *data position in cache* (Pos) information. As cache line size (cls) is usually larger than data size (ds), different array elements may fall into different positions in a cache line. We number the positions Pos from 0 to $\frac{cls}{ds} - 1$. The relation between dependence distance $\vec{v}\,^{\mathsf{Ref}_{\mathsf{snk}}}_{\mathsf{Ref}_{\mathsf{src}}}$ and dependence distance at specific source/sink positions $\vec{v}\,^{\mathsf{Ref}_{\mathsf{snk}},\mathsf{Pos}_{\mathsf{snk}}}_{\mathsf{Ref}_{\mathsf{src}},\mathsf{Pos}_{\mathsf{src}}}$ can be formulated as the following:

$$\vec{v}\,^{\mathsf{Ref}_{\mathsf{snk}},\mathsf{Pos}_{\mathsf{snk}}}_{\mathsf{Ref}_{\mathsf{src}},\mathsf{Pos}_{\mathsf{src}}} = \vec{v}\,^{\mathsf{Ref}_{\mathsf{snk}}}_{\mathsf{Ref}_{\mathsf{src}}} + (0, .., 0, \mathsf{Pos}_{\mathsf{snk}} - \mathsf{Pos}_{\mathsf{src}})$$

Note that this calculation assumes the innermost loop accesses the array continuously and $\vec{v}\,^{\mathsf{Ref}_{\mathsf{snk}}}_{\mathsf{Ref}_{\mathsf{src}}}$ quantifies the iteration different between $\mathsf{Pos}_{\mathsf{src}} = 0$ and $\mathsf{Pos}_{\mathsf{snk}} = 0$. Otherwise, it will need more sophisticated mathematics.

Then the shortest dependence distance for source reference $\mathsf{Ref}_{\mathsf{src}}$ in position $\mathsf{Pos}_{\mathsf{src}}$ can be formulated as the minimum distance among all possible sink references $\mathsf{Ref}_{\mathsf{snk}}$ and cache line positions $\mathsf{Pos}_{\mathsf{snk}}$.

$$\overrightarrow{sv}_{\mathsf{Ref}_{\mathsf{src}},\mathsf{Pos}_{\mathsf{src}}} = \min_{\forall \mathsf{Ref}_{\mathsf{snk}},\mathsf{Pos}_{\mathsf{snk}}} (\vec{v}\,^{\mathsf{Ref}_{\mathsf{snk}},\mathsf{Pos}_{\mathsf{snk}}}_{\mathsf{Ref}_{\mathsf{src}},\mathsf{Pos}_{\mathsf{src}}})$$

With the shortest distance, the reuse interval RI associated with it can be derived by multiplying $\overrightarrow{sv}$ with the number of references per iteration.

Inspirations for synthesis. Though we find the relation between reuses and dependences, calculating reuse intervals requires precise dependence distance vectors. Unfortunately, it is hard to derive the distance vectors when the loop structure is complex. Even for these two indices i and $2 * i$, we can not summarize the distance as a constant for all iterations. But for a specific i iteration, we always calculate a distance as a constant and then find a representation to summarize all iterations.

With the above observation and the equations for $\overrightarrow{sv}$, we identify factors that determine a RI at a specific iteration. The following four factors are sufficient to locate a specific access/reuse/RI for a given loop:

1. The symbolic loop bounds $\vec{\mathsf{B}}$.

2. The data position in cache $\mathtt{Pos}_{\mathtt{src}}$.
3. The source reference $\mathtt{Ref}_{\mathtt{src}}$.
4. The source iteration $\vec{\mathtt{I}}_{\mathtt{src}}$.

Because (1) The values of symbolic bounds $\vec{\mathtt{B}}$ determine the number of memory accesses in each loop, determining the memory access trace in element-granularity for a given loop. (2) The data position in cache $\mathtt{Pos}_{\mathtt{src}}$ reflects the data alignment. Combined with $\vec{\mathtt{B}}$, the cache-line-granularity memory access trace is fixed. (3) The source reference $\mathtt{Ref}_{\mathtt{src}}$ and its source iteration $\vec{\mathtt{I}}_{\mathtt{src}}$ locate a specific access to an array element. For a given trace, locating one access also finds its forward reuse interval.

The last three factors above are all for the source reference. We can have a dual form with sink reference if we are locating backward reuse intervals.

1. The data position in cache $\mathtt{Pos}_{\mathtt{snk}}$
2. The sink reference $\mathtt{Ref}_{\mathtt{snk}}$.
3. The sink iteration $\vec{\mathtt{I}}_{\mathtt{snk}}$.

$\mathtt{Pos}_{\mathtt{snk}}$ encodes data alignment as $\mathtt{Pos}_{\mathtt{src}}$ does. $\mathtt{Ref}_{\mathtt{snk}}$ and $\vec{\mathtt{I}}_{\mathtt{snk}}$ locates the second access in a reuse. This also indicates the $\vec{sv}$ can be formulated in the other way around from the forward $\vec{sv}_{\mathtt{Ref}_{\mathtt{src}},\mathtt{Pos}_{\mathtt{src}}}$ to backward $\vec{sv}_{\mathtt{Ref}_{\mathtt{snk}},\mathtt{Pos}_{\mathtt{snk}}}$.

With factors identified, the problem of finding a f can be formulated as finding a program PROG written in a Domain-Specific Language (DSL) with the above factors as variables.

3 Specifications and Their Structured Input-output Examples

To automatically find a PROG for a reuse interval, we need specifications to describe the program's structure and input-output examples for the synthesizer to describe the program's behavior. This paper focuses on finding the relation between reuses and the loop bounds with fixed alignment. So we do not add Pos to the variables in PROG.

3.1 Specifications

We propose three specifications for a program PROG that describes a reuse interval RI in the following format:

$$\mathtt{RI}_{\mathtt{identifier=constant}} = \mathtt{PROG}(\mathtt{variables})$$

A `identifier` locates a specific reuse interval by providing concrete values for the factors that determine a reuse interval. `variables` list all variable symbols that are allowed to appear in the program PROG.

Spec-src*: source-only.* This specification is trying to capture the relation between reuse interval RI and loop bound variables $\vec{\mathtt{B}}$ only by the factors related to the source reference. As Equation 1 shows, source reference id c_0 and a specific iteration $\vec{c}_1$ locates a reuse interval. Program's variables include all symbols from the source iteration vector and loop bound variables.

$$\mathtt{RI}_{\mathtt{Ref}_{\mathtt{src}}=c_0,\vec{\mathtt{I}}_{\mathtt{src}}=\vec{c}_1} = \mathtt{PROG}(\vec{\mathtt{I}}_{\mathtt{src}},\vec{\mathtt{B}}) \tag{1}$$

Spec-src-snk*: source-sink.* This specification adds one more factor, the sink reference $\mathtt{Ref}_{\mathtt{snk}}$, than Spec-src as Equation 2 shows. The variable list takes the same set of symbolic variables as the programs specified by Spec-src.

$$\mathtt{RI}_{\mathtt{Ref}_{\mathtt{src}}=c_0,\vec{\mathtt{I}}_{\mathtt{src}}=\vec{c}_1,\mathtt{Ref}_{\mathtt{snk}}=c_2} = \mathtt{PROG}(\vec{\mathtt{I}}_{\mathtt{src}},\vec{\mathtt{B}}) \tag{2}$$

As different $\vec{\mathtt{B}}$ values may change the sink reference of a reuse even if we fix $\mathtt{Ref}_{\mathtt{src}}$ and $\vec{\mathtt{I}}_{\mathtt{src}}$. Fixing $\mathtt{Ref}_{\mathtt{snk}}$ reduces the diversity of the values of a RI. This specification simplifies the task of finding a program to capture reuses from a single source to all candidate sink references to a specific sink reference.

Spec-src-snk+*: source-sink-enhanced.* This specification uses the same factors as Spec-src-snk does to locate a reuse interval. But it further suggests to use additional symbols $\vec{\mathtt{I}}_{\mathtt{snk}}$ in the variable list for the PROG as Equation 3 shows. $\vec{\mathtt{I}}_{\mathtt{src}}$ and $\vec{\mathtt{I}}_{\mathtt{snk}}$ together enable the opportunity to discover a PROG calculates the reuse interval through iteration difference $\vec{\mathtt{I}}_{\mathtt{snk}} - \vec{\mathtt{I}}_{\mathtt{src}}$.

$$\begin{aligned}\mathtt{RI}_{\mathtt{Ref}_{\mathtt{src}}=c_0,\vec{\mathtt{I}}_{\mathtt{src}}=\vec{c}_1,\mathtt{Ref}_{\mathtt{snk}}=c_2} &= \mathtt{PROG}(\vec{\mathtt{I}}_{\mathtt{src}},\vec{\mathtt{I}}_{\mathtt{snk}},\vec{\mathtt{B}})\\ \vec{\mathtt{I}}_{\mathtt{snk}} &= \mathtt{PROG}'(\vec{\mathtt{I}}_{\mathtt{src}},\vec{\mathtt{B}})\end{aligned} \tag{3}$$

Specification for f. With the above specifications for a specific RI, we can define f as a collection of programs for RIs from all references in all iterations to form a reuse interval histogram RIH. With Spec-src, the RIH can be defined as:

$$\forall \mathtt{Ref}_{\mathtt{src}} \in R, \vec{\mathtt{I}}_{\mathtt{src}} \in IS: \ \mathtt{RI}_{\mathtt{Ref}_{\mathtt{src}},\vec{\mathtt{I}}_{\mathtt{src}}}(\vec{\mathtt{B}}).$$

And with Spec-src-snk or Spec-src-snk+, the RIH is

$$\forall \mathtt{Ref}_{\mathtt{src}}, \mathtt{Ref}_{\mathtt{snk}} \in R, \vec{\mathtt{I}}_{\mathtt{src}} \in IS: \ \mathtt{RI}_{\mathtt{Ref}_{\mathtt{src}},\vec{\mathtt{I}}_{\mathtt{src}},\mathtt{Ref}_{\mathtt{snk}}}(\vec{\mathtt{B}}).$$

R is the set of static references in a program, and IS is the iteration space of the source reference. For a loop code with n references and t iterations, RIH will have a collection of $O(nt)$ programs with Spec-src and $O(n^2t)$ programs with Spec-src-snk, Spec-src-snk+. n is a fixed number for a given program. But t changes with the $\vec{\mathtt{B}}$ values.

Specification of RIH is iteration-based. It reveals the reuse intervals at each iteration. But it is impossible to collect the full set of iterations by enumerating all possible $\vec{\mathtt{B}}$ values. To avoid enumerating, we need an approach to generalize the RIH from a subset of $\vec{\mathtt{B}}$ values to all other possible $\vec{\mathtt{B}}$ values. To achieve this goal, we first characterize how the shape of iteration space changes with $\vec{\mathtt{B}}$ in ①. Then, we redistribute the iterations from one iteration space to another by rewriting in ②.

① *Characterizing iteration space:* The lower and upper bound expressions of a loop determine the shape of an iteration space. As Figure 3a shows, the lower bound f_L and upper bound f_U are functions of induction variables from outer k loops $\vec{\mathtt{I}}_{0\sim k-1}$ and bounds $\vec{\mathtt{B}}$ from the input. The presence of induction variables from the output loop will make shape analysis hard as the shape may change with different outer loops' induction variables. If the current loop has two outer loops and each has a thousand iterations, the number of different upper bounds and lower bounds may be one million. In this case, to find a program for f_L or f_U will need to generate an input-output example with one million records.

```
for (int i = $f_L(\vec{I}_{0\sim k-1}, \vec{B})$; i ≤ $f_U(\vec{I}_{0\sim k-1}, \vec{B})$;  i++) {
    ...
}
```

(a) Original loop

```
$f'_L(\vec{B}) = \min_{\forall \vec{I}_{0\sim k-1}}(f_L(\vec{I}_{0\sim k-1}, \vec{B}))$;
$f'_U(\vec{B}) = \max_{\forall \vec{I}_{0\sim k-1}}(f_U(\vec{I}_{0\sim k-1}, \vec{B}))$;
for (int i = $f'_L(\vec{B})$; i ≤ $f'_U(\vec{B})$; i++) {
    if ($f_L(\vec{I}_{0\sim k-1}, \vec{B})$ ≤ i && i ≤ $f_U(\vec{I}_{0\sim k-1}, \vec{B})$) {...}
}
```

(b) Loop after rectangularization

Figure 3. Iteration space rectangularization

To efficiently characterize the iteration space, we project iteration space to a rectangular shape, as Figure 3b shows. Loop bounds will now be the minimal value of f_L and maximum value of f_U among all $\vec{I}_{0\sim k-1}$ values as f'_L and f'_U shows. An if statement will be added to preserve the semantics of the original loop. The transformed loop will have the same access traces as the original one by adding dummy iterations that do not generate access. With all inner loops rectangularized, we move all the if statements to the inner-most loop to get a rectangular iteration space for nested loops.

Rectangularization makes the shape of iteration space independent of outer loops' induction variables. The iteration space only changes with $\vec{B}$. Its specification is listed as follows:

$$\forall loop, L_{loop} = \texttt{PROG}_{min}(\vec{B}), U_{loop} = \texttt{PROG}_{max}(\vec{B})$$

We can find two programs for each loop in the program to describe its lower and upper bounds.

② *Redistributing source iterations:* Here we assume that the redistributing happens from a smaller iteration space to a larger iteration space. The assumption makes sense as we would like to find symbolic RIH with a set of small $\vec{B}$ values to predict the reuses for a larger $\vec{B}$ value. As Figure 4 shows, the two iterations marked by blue dots in (a) will be clustered without redistributing in (b). While, redistribution will uniformly fill the larger iteration space with the iterations from smaller iteration space in (c).

Redistribution for the i^{th} loop can be performed by the following equation.

$$\vec{I}'_{\texttt{src},i} = \frac{U_i(\vec{b}_1) - L_i(\vec{b}_1)}{U_i(\vec{b}_0) - L_i(\vec{b}_0)} \times (\vec{I}_{\texttt{src},i} - L_i(\vec{b}_0)) + L_i(\vec{b}_1)$$

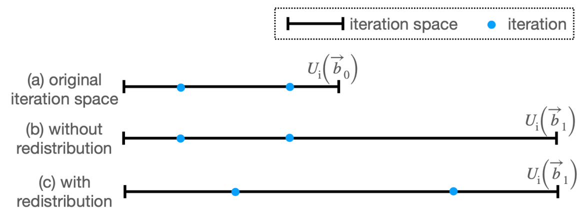


Figure 4. Redistributing iterations for one-dimensional iteration space

Table 1. Input-output examples for Spec-src

input	$\vec{I}_{\texttt{src}}$	$\vec{B}$	output	RI
value	$\vec{c}_1$	$\vec{b}_0$		ri_0
	$\vec{c}_1$	$\vec{b}_1$		ri_1
	...	...		...
	$\vec{c}_1$	$\vec{b}_{n-1}$		ri_{n-1}

select records where $\texttt{Ref}_{\texttt{src}} = c_0, \vec{I}_{\texttt{src}} = \vec{c}_1$

$\vec{I}_{\texttt{src},i}$ will be replaced by $\vec{I}'_{\texttt{src},i}$, which is calculated by summing the scaled distance from $\vec{I}_{\texttt{src},i}$ to lower bound calculated by $\vec{b}_0$ and the lower bound calculated with $\vec{b}_1$.

3.2 Structured input-output examples

With specifications defined, we then trace the program with different $\vec{B}$ values. The number of different $\vec{B}$ values decides the number of traces we have. For example, if we provide three train sizes 4, 8, 16 for each element of $\vec{B}$ and the length of $\vec{B}$ is 2, we will have $3^2 = 9$ traces. The $\vec{B}$ values will be (4, 4), (4, 8), (4, 16), (8, 4), (8, 8), (8, 16), (16, 4), (16, 8), and (16, 16).

Each trace records all reuses in an execution with the following information: (1) the reference ID $\texttt{Ref}_{\texttt{src}}$ and iteration $\vec{I}_{\texttt{src}}$ of a reuse source. (2) The same information $\texttt{Ref}_{\texttt{snk}}, \vec{I}_{\texttt{snk}}$ for its reuse sink. (3) Its reuse interval RI. The trace record format for each reuse is listed as the following:

$$[\texttt{Ref}_{\texttt{src}}, \vec{I}_{\texttt{src}}, \texttt{Ref}_{\texttt{snk}}, \vec{I}_{\texttt{snk}}, \vec{B}, \texttt{RI}]$$

From collected traces, we extract and construct input-output examples for each specification with different formats. Here we assume there are n different values for $\vec{B}$.

***Examples for* Spec-src.** With n traces, we extract the records whose $\texttt{Ref}_{\texttt{src}} = c_0$ and $\vec{I}_{\texttt{src}} = \vec{c}_1$ to form the examples for $\texttt{RI}_{\texttt{Ref}_{\texttt{src}}=c_0, \vec{I}_{\texttt{src}}=\vec{c}_1}$ as Table 1 shows. The input symbols are $\vec{I}_{\texttt{src}}$, $\vec{B}$, and the output symbol is RI. All records share the same source iteration, so all values for $\vec{I}_{\texttt{src}}$ in the input-output examples are the same. The values for $\vec{B}$ will be all different as we collect traces with different bound values.

***Examples for* Spec-src-snk.** These examples share the input and output symbols with the examples for Spec-src. But the records are selected with an additional restriction for

Table 2. Input-output examples for `Spec-src-snk`

input	$\vec{\mathtt{I}}_{\mathtt{src}}$	$\vec{\mathtt{B}}$	output	$\mathtt{RI}$
value	$\vec{c}_1$	$\vec{b}_0$		ri_0
	$\vec{c}_1$	$\vec{b}_1$		ri_1
	...	...		...
	$\vec{c}_1$	$\vec{b}_{n-1}$		ri_{n-1}

select records where $\mathtt{Ref}_{\mathtt{src}} = c_0, \vec{\mathtt{I}}_{\mathtt{src}} = \vec{c}_1, \mathtt{Ref}_{\mathtt{snk}} = c_2$

Table 3. Input-output examples for `Spec-src-snk+`

input	$\vec{\mathtt{I}}_{\mathtt{src}}$	$\vec{\mathtt{I}}_{\mathtt{snk}}$	$\vec{\mathtt{B}}$	output	$\mathtt{RI}$
value	$\vec{c}_1$	$\vec{i}_{\mathtt{snk},0}$	$\vec{b}_0$		ri_0
	$\vec{c}_1$	$\vec{i}_{\mathtt{snk},1}$	$\vec{b}_1$		ri_1
	...	...	...		...
	$\vec{c}_1$	$\vec{i}_{\mathtt{snk},n-1}$	$\vec{b}_{n-1}$		ri_{n-1}
input	$\vec{\mathtt{I}}_{\mathtt{src}}$	$\vec{\mathtt{B}}$		output	$\vec{\mathtt{I}}_{\mathtt{snk}}$
value	$\vec{c}_1$	$\vec{b}_0$			$\vec{i}_{\mathtt{snk},0}$
	$\vec{c}_1$	$\vec{b}_1$			$\vec{i}_{\mathtt{snk},1}$
	...	...			...
	$\vec{c}_1$	$\vec{b}_{n-1}$			$\vec{i}_{\mathtt{snk},n-1}$

select records where $\mathtt{Ref}_{\mathtt{src}} = c_0, \vec{\mathtt{I}}_{\mathtt{src}} = \vec{c}_1, \mathtt{Ref}_{\mathtt{snk}} = c_2$

$\mathtt{Ref}_{\mathtt{snk}}$. Table 2 shows the input-output examples generated for $\mathtt{RI}_{\mathtt{Ref}_{\mathtt{src}}=c_0, \vec{\mathtt{I}}_{\mathtt{src}}=\vec{c}_1, \mathtt{Ref}_{\mathtt{snk}}=c_2}$. Note that the values for `RI` may be different from the values in examples in `Spec-src` as the reuse sink may not be c_2.

***Examples for* `Spec-src-snk+`.** These examples extract the same set of records as `Spec-src-snk` does but structure them in a different format. It contains two input-output examples, one for `RI` and the other is for $\vec{\mathtt{I}}_{\mathtt{snk}}$. The examples for `RI` contain one additional input symbol $\vec{\mathtt{I}}_{\mathtt{snk}}$ which is consistent with its specification.

***Sparsity of values for** RI.* For some $\vec{\mathtt{B}}$ values, there may be no reuse for the selected source iteration. Therefore, we will assign 0 for `RI` in this case. 0 entries are essential in the input-output examples as, together with non-zero entries, it will indicate the conditions which decide when reuse will exist.

Examples for iteration space. For each loop, we would like to discover two programs that describe the lower and upper bounds. These examples only take $\vec{\mathtt{B}}$ as the input and L or U as output. The number of records in each example is equal to the number of different $\vec{\mathtt{B}}$ values. The values for outputs are the minimal or the maximum of all values of induction variable l.

Table 4. Input-output examples for iteration space of a loop with induction variable l

input	$\vec{\mathtt{B}}$	output	L
value	$\vec{b}_0$		$\min(\forall l_0)$
	$\vec{b}_1$		$\min(\forall l_1)$
	...		...
	$\vec{b}_{n-1}$	.	$\min(\forall l_{n-1})$
input	$\vec{\mathtt{B}}$	output	U
value	$\vec{b}_0$		$\max(\forall l_0)$
	$\vec{b}_1$		$\max(\forall l_1)$
	...		...
	$\vec{b}_{n-1}$		$\max(\forall l_{n-1})$

4 Synthesis Framework

4.1 Unification search with elimination-free DSL

With generated structure input-output examples, we adopt the syntax-guided search with unification [2, 4, 37].

Search with unification. Algorithm 1 demonstrates the search process. We first try to find a program that can satisfy all input-output examples with `BottomUpSearch` in Line 2. If a program is not found by `BottomUpSearch`, we then split the input-output examples to two sets: left L and right R in Line 4. Also, we will generate a predict set P, which indicates whether the original examples go to L or R by replacing original outputs with Boolean values. For P, we perform a bottom-up search in Line 5 to learn a Boolean expression for the condition of an if-then-else statement (`ITE`). For L and R, we recursively call `Unification` function in Line 6 and 7 to learn an `PROG` that saftisfy L and R separately. If all programs are found for sets P,L and R, we return the `ITE` program in Line 9. If not, we do backtracking to Line 4 with different splitting strategies. If all failed, return not found.

For `SplitExamples()`, we design three different splitting methods:

1. **Zero:** If 0 presents in the outputs, we group the examples with 0 outputs to L and others to R.
2. **Freq:** Find the most frequent output. Group the most frequent ones to L and others to the R.
3. **Half:** Sort the output values and split the examples to half and half.

Zero rule is to separate the 0 with other `RI` values. It helps to learn a condition where reuse happens by grouping all 0s. **Freq** and **Half** rules are to reduce the diversity of the output values. The less diverse the outputs are, the more likely to find a shorter `PROG`.

`BottomUpSearch()` uses an iterative algorithm to gradually grow a set of programs by following a DSL's syntax. It first initializes the set of candidate programs in line 17. The initial candidates usually contain variables and small

Algorithm 1: Unification algorithm

```
1  Function Unification(examples):
2      PROG = BottomUpSearch (examples);
3      if PROG not found then
4          [L, R, P] = SplitExamples (examples);
5          $PROG_P$ = BottomUpSearch (P);
6          $PROG_L$ = Unification (L) ;
7          $PROG_R$ = Unification (R) ;
8          if $PROG_P$, $PROG_L$, $PROG_R$ are found then
9              PROG = ITE $PROG_P$ $PROG_L$ $PROG_R$ ;
10         else
11             backtracking;
12         end
13     end
14     return PROG;
15 End
16 Function BottomUpSearch(examples):
17     init pList;
18     while CheckCorrect(pList) do
19         pList = Grow(pList);
20         pList = EliminiateEquivalents(pList);
21     end
22     return GetCorrect(pList);
23 End
```

constants. Then, it checks and grows the program set iteratively by a while loop in lines 18- 21. CheckCorrect() scans all the programs in the *pList* to see whether there is a program that satisfies all input-output examples. If yes, the loop terminates and BottomUpSearch() returns the program in line 22. If not, Grow() will iterate throw all programs in *pList* and all operations in DSL to construct new programs. The constructed programs may be equivalent to each other, so in line 20 we only keep one program among all its equivalents. [1] Note that the number of candidate programs is infinite. We usually set a time-bound for this growing process.

Elimination-free DSL. The language we defined for synthesis is a simple language for integer expressions that support branches. The language contains constant numbers Nums; a set of integer variables Vars; integer operations addition $+$, multiplication $\times$, if-then-else ITE; Boolean operations negation $\neg$, and $\wedge$, less than $<$. It is sufficient to cover all possible integer functions to describe a reuse interval and describe the conditions to partition the iteration space. Besides coverage, avoiding generating equivalent programs in different forms is also a concern for efficiency when defining the language. We avoid adding inverse operations such as $-$, $\div$, and $\vee$. We also attach a generation and a lexical order to each program to carefully control the program construction to avoid generating equivalent programs.

[1]Searching for an integer program with an elimination-free DSL will skip EliminiateEquivalents()

Figure 5 shows the enforced structure for an elimination-free DSL. For Nums, the initial set of numbers are all prime numbers with generation assigned 0. Multiplication is the only method to construct new numbers. Generation constraint $g' - 1 = max(g0, g1)$ only allows program construction by using at least one program from previous generation.

For Times expressions, we do not allow Plus expressions on both sides of a Times expression due to distributive law. Only Vars and Nums can serve as operands for a Times expression. A Num can only appear on the left-hand side of a nested Times expression. As multiplication is commutative and associative, we can always move multiple Nums in a Times expression to the left and apply multiplication to get a single Num. For Vars, we enforce a lexical order to avoid generating equivalents by reordering the Vars. For Plus, Num can only appear once on the left-hand side. The right-hand side is a nested sum of Times expressions.

The lexical order *lex* is defined as a vector. Each position records the number of the appearance of a Var in the expression. The vector length equals to the number of different Vars defined in DSL.

- *lex* of any Num is a all-zero vector.
- *lex* of Var is a vector with the corresponding element equals to 1.
- *lex* of Times and TimesRight is a vector defined by the sum of *lex* of all Vars in it.
- *lex* of Plus and PlusRight returns the *lex* of the left-hand side expression.

For Boolean expressions, And is also commutative and associative. We design a similar structure as Plus does by only expanding the right-hand side. Lexical orders are defined to reserve one sorted expression among all its variations.

- *lex* of And returns the *lex* of the left-hand side expression.
- *lex* of Not returns the *lex* of its containing And expression.
- *lex* of Lt is defined as appending the lex of the left-hand side expression and right-hand side expression. Note that when comparing the *lex* between Lt and other expressions will be automatically padding to the same length by 0s.

For Not and Lt, generation alone can guarantee elimination-free as they are not commutative. We choose not to attach a Not expression to Lt as the negation a Lt expression can be generated by exchanging left-hand and right-hand side expressions and adding 1 to the right. [2]

[2]Note that the two IntExpr expressions should not share common factors/terms that can be canceled.

$$\begin{aligned}
\texttt{IntExpr} &= \texttt{Num} \mid \texttt{Var} \mid \texttt{Plus} \mid \texttt{Times} \\
\texttt{Num}^{g'} &= \texttt{Num}^{g0} \times \texttt{Num}^{g1} \\
\texttt{Times}^{g'} &= (\texttt{Num} \mid \texttt{Var})^{g0,\texttt{lex0}} \times \texttt{TimesRight}^{g1,\texttt{lex1}} \\
\texttt{TimesRight}^{g'} &= \texttt{Var}^{g0,\texttt{lex0}} \times (\texttt{Var} \mid \texttt{TimesRight})^{g1,\texttt{lex1}} \\
\texttt{Plus}^{g'} &= (\texttt{Num} \mid \texttt{Var} \mid \texttt{Times})^{g0,\texttt{lex0}} + \texttt{PlusRight}^{g1,\texttt{lex1}} \\
\texttt{PlusRight}^{g'} &= (\texttt{Var} \mid \texttt{Times})^{g0,\texttt{lex0}} \\
&\quad + (\texttt{Var} \mid \texttt{Times} \mid \texttt{PlusRight})^{g1,\texttt{lex1}}
\end{aligned}$$

$$\begin{aligned}
\texttt{Lt}^{g'} &= \texttt{IntExpr}^{g0} < \texttt{IntExpr}^{g1} \\
\texttt{Not}^{g'} &= \neg \texttt{And}^{g0} \\
\texttt{And}^{g'} &= \texttt{Lt}^{g0,\texttt{lex0}} \wedge (\texttt{And} \mid \texttt{Not} \mid \texttt{Lt})^{g1,\texttt{lex1}}
\end{aligned}$$

$$Constraints: \begin{cases} \texttt{Num}^{0} \in \texttt{prime numbers} \\ g' - 1 == max(g0, g1) \\ \texttt{lex0} < \texttt{lex1} \end{cases}$$

Figure 5. Elimination-free DSL

4.2 Expectations/biases of the PROGs

Though elimination-free DSL only generates one program among all its equivalent forms in the domain of $\mathbb{Z}$ for all its input variables. We may still find multiple programs that satisfy all input-output examples, as inputs from the examples can not cover all possible integers. Thus, we add expectations/biases to specify the preferred program and prune the search space.

Reuse-type inferred forms. For a specific iteration, there are two different types of reuse intervals. One is *constant* reuse interval, that the non-zero RI values for different $\vec{B}$s do not change. The other is *scaling* reuse interval, that the non-zero RI values for different $\vec{B}$s scale with $\vec{B}$ or $\vec{I}_{src}$.

For constant RI, the preferred program for it is a Num. For scaling RI, the preferred program should contain $\vec{B}$ or $\vec{I}_{src}$ for Spec-src, Spec-src-snk, and $\vec{I}_{snk}$ for Spec-src-snk+. When two programs that both satisfy all input-output examples during the search, we choose the program based on the following priorities:

IntExpr: $P(\texttt{Num}), P(\vec{I}_{snk}) < P(\vec{I}_{src}), P(\vec{B})$

We assign higher priority for programs with variables than constant programs with Nums only. With our examples from different $\vec{B}$ values, the chance of having the same RI value in an input-output example for scaling RI is small.

If a RI scales with $\vec{I}_{src}$, the chance to successfully summarize it in $\vec{B}$ is small as in the examples, the values for $\vec{I}_{src}$ are the same for all records but $\vec{B}$ values differ from each other. For $\vec{B}$ and $\vec{I}_{src}$, if the coefficients of indices of source and sink references' $\vec{I}_{src}$ are different or loop bounds are functions of $\vec{I}_{src}$s. The RI is more likely to scale with $\vec{I}_{src}$, otherwise we prefer $\vec{B}$. $\vec{I}_{snk}$ only appears in programs under Spec-src-snk+. As $\vec{I}_{snk} - \vec{I}_{src}$ indicates dependence distance, we prefer they appear in pairs to assign higher priorities than Nums.

Code-structure inferred forms. Instead of enumerating all elimination-free programs, the code structure can help to reduce the search space.

- Avoid searching with all bound symbols in $\vec{B}$. As reuses can only happen between static references to the same array. For a source reference, only the accesses between the access from source reference and the access from its sink reference matter. All loop iterations that do not overlapping with this path of will be irrelevant. We can safely ignore the bound symbols only for the loops that before loop containing the source reference and after the loop containing the sink reference.
- Bounding the structures of Times expressions. As reuse interval can be calculated as iteration difference times the number of accesses per-iteration. For the loops that the number of memory accesses can be easily summarized in expressions, we explicitly added them as biases for Times expressions. For the loops that we can not infer a bias, we can still bound the exponent of Times to a small number, such as the depth of the loops.
- Limiting the value range for Nums. The final expected IntExpr programs should be linear combinations of Times expressions. The coefficients should be small numbers, whose absolute values are less equal to the maximum number of static references in a loop among all loops.

4.3 Complexity

The complexity of the unification algorithm depends on the number of bottom-up searches performed and its complexity.

For BottomUpSearch(), we assume pList has n IntExpr programs and m BoolExpr programs. The number of IntExpr programs can be generated is $O(n^2)$ and the number of BoolExpr programs can be generated is $O(m^2 + n^2)$ in one generation. Though programs grow exponentially, pruning with provided bias can slow down the growth.

5 Evaluation

This section first demonstrates the synthesized symbolic reuse intervals, the derived histograms, and miss ratios for the 5-point stencil program. Then, we evaluate the synthesizer with PolyBench/C 4.2.1 [28]. It contains 30 numerical kernels extracted from linear algebra, image processing, physics simulation, dynamic programming and statistics applications. [3]

5.1 Case study with 5-point stencil

We generate the input-output examples for stencil in Figure 2a from its traces with B0 equals 4, 8, 10, 12, 20.

Symbolic RI. For source reference $a[i+1][j]$, the reuses may happen at sink references $a[i+1][j]$, $a[i][j]$ and $a[i][j+1]$.

For accesses that happen at $a[i+1][j]$ and reused at $a[i+1][j]$, their reuses will happen at next j iteration with reuse interval equals 6. The programs we found for all specifications agree with this analysis. With `Spec-src-snk+`, we will find the following program if we force the synthesizer to find a program with $\vec{\text{I}}_{\text{src}}$ and $\vec{\text{I}}_{\text{snk}}$. It is equivalent to 6.

```
RI =       (6 × (Isnk1 - Isrc1))
Isnk1 =    (if (B < j) then 0 else (1 + Isrc1))
```

For accesses that happen at $a[i+1][j]$ and reused at $a[i][j+1]$, their reuse intervals diverse and range from 93, 99, 105 for B = 20. The programs we found are also different.

```
RI =  ((5 * B) - 7)
      ((5 * B) - 1)
      ((6 * B) - 15)
```

If we are using elimination-free `DSL` without subtraction, we can find it's equivalent form with addition for the same input-output examples, such as $((4 * B) + 13)$ for ((5 * B) - 7).

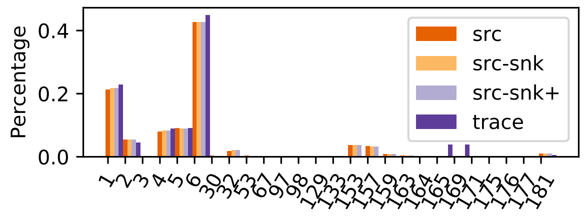
Tiling. We rewrite the 5-point stencil by adding two additional inner loops to tile the iteration space by TS × TS. The synthesis process is no different but from one variable B to two variables by adding TS. Now we can find reuse intervals with TS, such as $(7 + (TS + B))$.

Growing speed. Elimination-free DSL and bias play an essential role in reducing search space. It decides whether we can successfully find a program within a short amount of time. Table 5 shows the number of programs constructed with a baseline `DSL` and its elimination-free form in the first three generations. They start with the same set of programs in the first generation. The number of programs grows exponentially with the baseline `DSL` but linearly with elimination-free `DSL`.

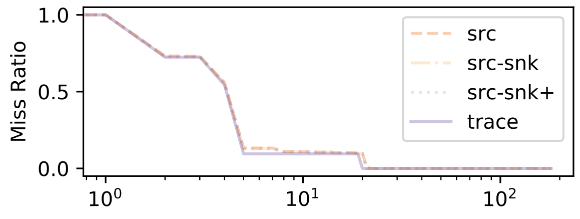
Table 5. Programs grow speeds of a baseline DSL with `Plus`, `Times` and its elimination-free form

Generation	1	2	3
DSL (PLUS, TIMES)	16	287	84314
Elimination-free DSL	16	47	129

[3]We open-sourced our tool and all synthesized symbolic RIs for PolyBench in https://github.com/xxx (link removed for double-blind review).



(a) Reuse interval histogram



(b) Miss ratio curve

Figure 6. Prediction with B = 32 by symbolic `RIH` specified by `Spec-src`, `Spec-src-snk` and `Spec-src-snk+`

***Predicting with* `RIH`.** By summarizing synthesized `RI`s, we get symbolic `RIH`s for all three specifications. Figure 6a shows the predicted reuse interval histograms from all three `RIH`s by assigning B to 32 and the trace. The x-axis shows the values of each reuse interval. All `RIH`s can capture short reuse intervals 1 to 6 and the largest reuse interval 181 with very small percentage errors. For reuse intervals 165 and 169, the predicted reuse intervals are smaller. There are two possible reasons: (1) the program we found has other equivalent forms that can produce larger values. (2) when the bound changes, the cache-line granularity has different reuses patterns offset by around two inner iterations (12 accesses).

Figure 6b shows the miss ratio curves derived from each histogram by average eviction time [16] for LRU fully associative cache. [4] All three miss ratio curves almost overlap with traced miss ratio curves.

5.2 Examples for PolyBench

We choose five train sizes, 4, 6, 8, 12, 20, for each bound in all benchmarks. Table 6 shows the size of generated structured input-output examples for all benchmarks.

The number of symbolic bounds $\texttt{len}(\vec{\text{B}})$ ranges from 1 to 5. $\texttt{len}(\vec{\text{B}})$ and the number of train sizes provided determines the number of records in each input-output example. For example, there are 5^1 records for programs with one symbolic bound and 5^5 records for programs with five symbolic bounds.

All three specifications share the same shape examples. The number of references and induction variables in a program determines the number of files in shape examples. It

[4]The miss ratios can be derived for multi-level set-associated cache [40] and for parallelized loops [23] with `RI` distributions.

Table 6. The size of generated structured input-output examples with 20% sampling rate for cacheline size 32B and data size 8B

Name	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen
len($\vec{B}$)	4	5	2	2	2	1	2	2	2	3
#Shape	138	198	656	100	100	188	318	180	200	196
#RI$_{spec1}$	555	809	1364	129	128	90	418	258	304	1199
#RI$_{spec2/3}$	581	850	1415	136	132	85	432	270	304	1238
# $\vec{I}_{snk}$	1699	2508	4179	266	258	210	956	672	608	4785
Name	durbin	fdtd-2d	floyd-warshall	gemm	gemver	gesummv	gramschmidt	heat-3d	jacobi-1d	jacobi-2d
len($\vec{B}$)	4	5	2	2	2	1	2	2	2	3
#Shape	118	240	150	70	170	110	182	704	128	288
#RI$_{spec1}$	55	214	383	285	224	137	323	4088	112	612
#RI$_{spec2/3}$	53	334	378	296	223	139	343	4218	129	650
# $\vec{I}_{snk}$	106	668	1134	858	436	264	959	16872	258	1950
Name	lu	ludcmp	mvt	nussinov	seidel-2d	symm	syr2d	syrk	trisolv	trmm
len($\vec{B}$)	4	5	2	2	2	1	2	2	2	3
#Shape	246	324	76	228	306	132	164	80	76	86
#RI$_{spec1}$	96	114	120	96	510	239	213	286	41	147
#RI$_{spec2/3}$	89	104	120	110	517	238	245	288	38	155
# $\vec{I}_{snk}$	258	248	240	220	1551	678	721	836	66	436

ranges from 76 (trisolv) to 704 (heat-3d). More loops and more references lead to more files, such as adi and heat-3d.

For all formats, the numbers of files generated (RI sampled) are similar. We sampled 20% of the source iterations when generating the input-output examples for each specification. Note that if the number of source iterations is still larger than 500 after the sampling, we further reduce the number of samples to 500. The number of examples for $\vec{I}_{snk}$ is proportional to #RI$_{spec3}$ by a factor of its vector length.

5.3 Overhead and precision

Different examples have different numbers of variables and output values. The actual running time will range from seconds to several minutes for each example. Each input-output example is independent, so we create a thread pool with 60 threads to launch synthesizer instances in parallel on a server with 64-core Intel(R) Xeon(R) CPU E7-4809 v3 @ 2.00GHz and 32G memory. It will take days to go through all the examples generated for all specifications. [5] Note that symbolic RI is only needed to be synthesized once for all future predictions. The synthesizing overhead can be amortized.

Figure 7 shows the predicted miss ratio curves when setting all bounds to 32 under different specifications for PolyBench. We choose 32 because (1) it enables fast tracing to generate a baseline to compare. (2) both small and large bounds reflect errors of symbolic reuse intervals. Larger bounds only add more iterations than the diversity of reuse interval values.

[5]We can further reduce this overhead to hours if we limit the number of samples for $\vec{I}_{src}$ to 200 for each benchmark/specification. Two hundred reuses will construct a histogram where each RI represents 0.5% of the reuses.

From the figure, we can see that all predicted curves capture the shape of the traced curve. Some of the predicted curves drop to 0 early, such as *2mm*, and *heat-3d*. It means the longest predicted reuse is smaller than the traced one. The following three cases will lead to this: (1) there exists a saw-tooth reuse pattern, such as accesses trace "abcddcba" with reuse intervals 7 for "a", 5 for "b", 3 for "c", and 1 for "d". A saw-tooth pattern will create a large number of scattered reuse intervals where no two reuse intervals share the same value. The constructed input-output examples may fail to include the longest reuse due to sampling. (2) the program we found may have lower coefficients or lower exponents. It scales to a smaller value with a new $\vec{B}$ value. (3) the synthesizer fails to find the program. For an input-output example with a large RI value, its program may need more generations to construct, which takes more time.

Comparing different specifications, we can see that in most cases, they have similar accuracy. Spec-src-snk performs slightly better than the other two specifictions for *3mm*, *correlation* and performs slightly worse for *2mm*, *bicg*, *covariance*. Specifications and the iterations sampled for each specification affect the performance. Spec-src-snk tends to have less diverse outputs in the examples than Spec-src, and Spec-src-snk+ has more variables during the search by adding the sink iterations than the other two.

Note that we can redo the searches for all failed or inaccuracy symbolic RIs for the examples/iterations with different search times or search biases. The new searched programs can update a subset of the programs in the RIH without side effects.

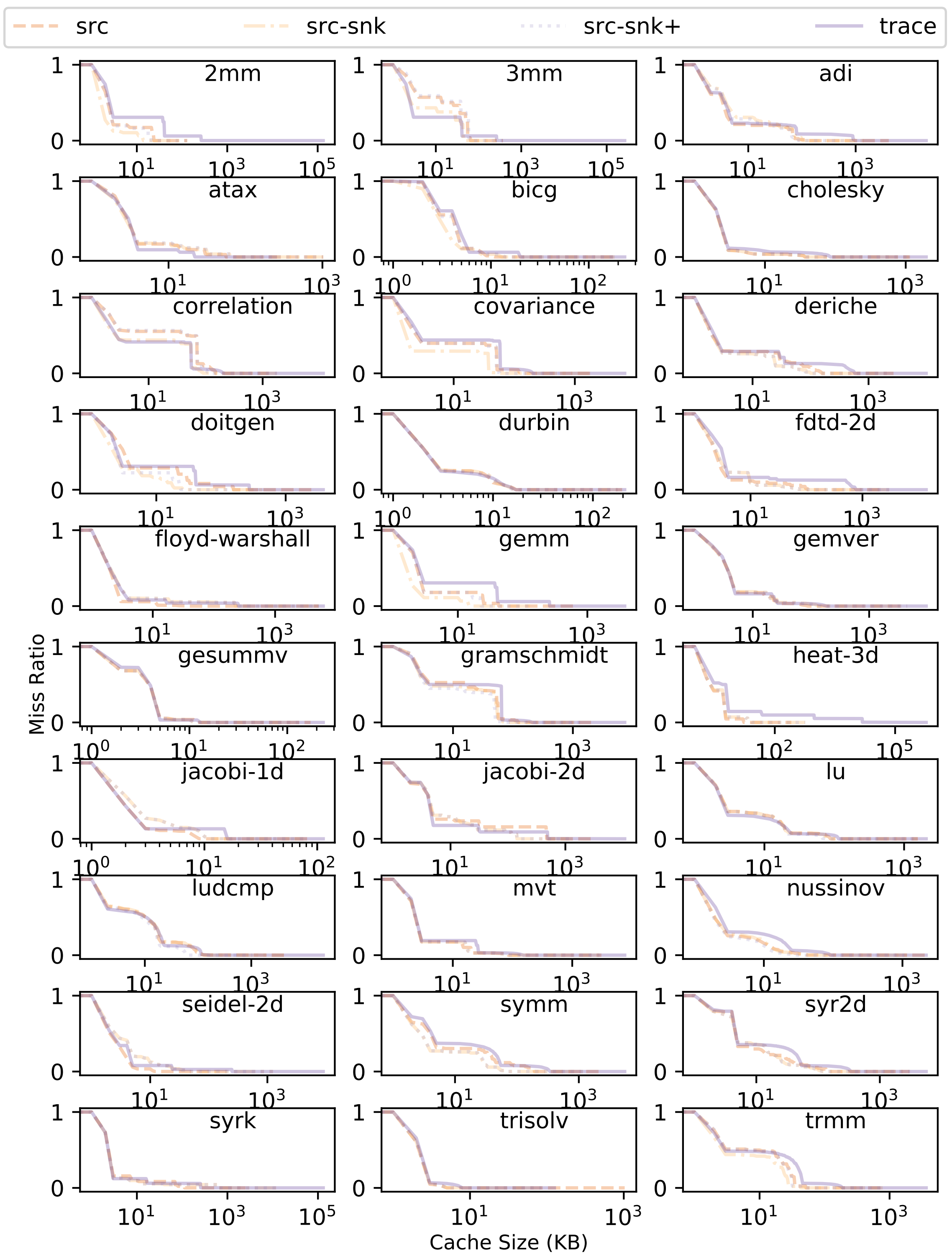


Figure 7. Miss ratio curves predicted by synthesized reuse intervals

6 Related work

Static locality analysis. Locality analysis is important to guide loop transformations, cache hint generation, and parallelism-locality trade-offs in compilers. A lot of researches have been exploring it, and the methods are becoming more and more sophisticated, from working set of loops [13, 19, 38] to reuse distance [7, 8] and reuse interval [10] of reuses:

Estimating the working set: Wolf and Lam used reuse vectors, which is derived from dependence distance, to calculate the number of memory accesses for the innermost L loops [38]. Kennedy and McKinley proposed loop cost functions to calculate the number of cache lines accessed by the references in the innermost loop [19]. Ferdinand et al. proposed an abstract interpretation approach to track the working set in a set-associated cache. [3] Touzeau et al. further improved it by introducing new abstractions [35, 36]. Ghosh et al. proposed cache miss equations to calculate misses from reuse vectors [38] for specific cache sizes, which is implemented in SUIF compiler framework [13]. Chatterjee et al. used Presburger formulas to express misses instead of using reuse vectors [9]. Bao et al. proposed an integer-set-based model to calculate misses of polyhedral programs for set-associative cache [5].

Estimating reuse distance/interval: Cascaval and Padua used dependence distance to derive a symbolic reuse distance histogram which can derive all cache size miss ratios [8]. Beyls and D'Hollander proposed the reuse distance equation based on the polyhedral model to derive a reuse distance histogram [7]. Gysi et al. proposed an efficient method to calculate reuse distance by combining symbolic counting and partial enumeration, which extends reuse distance equations to non-affine polynomials [15]. Chen et al. generate specialized loops to sample reuse intervals to construct a reuse interval histogram. [10]. Instead of using mathematically models [7, 8, 15], or sampler programs [10] to represent locality, our work generates iteration-based symbolic reuse intervals for locality.

Programming by examples. Providing examples are much simpler than writing concrete codes. It has been researched since 80s. The early systems are trying to capture repetitive patterns by pattern matching, such as Pygmalion [32], Tinker [22], Eager [11], Cima [25]. PBE can also be encoded as an inductive learning process by enumerative search [20]. It is often encoded with version space which learns a compact representation by provided general-to-specific relation [26] or first-order logic to find a set of rules to describe the input-output relation. Such as THESYS learns LISP looping structures [34], and FlashFill learns string processing programs [14]. Programming by examples (PBE) can simplify the programming interface for humans and enable an intelligent system that rewrites its code. Affine reconstruction of the loop takes memory access traces and can reconstruct all the static control parts in Polybench [30, 31]. Instead of reconstructing the original code, our work searches for the locality representation of code.

7 Conclusion and Future Work

This paper designs and implements the first input-output-examples-based synthesis system for locality analysis. It constructs and searches the candidate programs for reuse intervals with elimination-free DSL, specifications, and biases. The effectiveness of the system is demonstrated with a 5-point stencil and PolyBench.

This paper opens many future directions of work.

Beyond sequential scientific loops. Same as other static locality analysis approaches, this paper focuses on loop-based codes. To get locality information for a more complex program often requires trace analysis, such as programs containing alias, parallelism, or irregular accesses (sparse matrices). It requires additional efforts to encode characters of input data or thread schedulers to the input-output examples.

Program analysis through synthesis. In the past, static locality analysis requires humans to exam the code, visualize its execution in mind and build mathematical models. In this paper, we convert the static locality analysis problem as a synthesis problem. Similar to program sketching [33], which leaves the algorithmic details to the synthesizer, our tool offloads model details to the synthesizer. We may generalize this approach to other analyses.

Prescriptive cache management. The cache management in the past is reactive. Recent work shows the benefit of a prescriptive cache [21, 29]. This work provides symbolic iteration-based reuse information, which is essential for the fine-granularity prescription.

References

[1] [n.d.]. CUDA Runtime API–Memory Management. *NVIDIA* ([n. d.]). https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

[2] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings.* 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

[3] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache behavior prediction by abstract interpretation. In *International Static Analysis Symposium*. Springer, 52–66.

[4] Rajeev Alur, Pavol Černỳ, and Arjun Radhakrishna. 2015. Synthesis through unification. In *International Conference on Computer Aided Verification*. Springer, 163–179.

[5] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and Ponnuswamy Sadayappan. 2017. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–26.

[6] 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.

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

[8] Calin Cascaval and David A. Padua. 2003. Estimating cache misses and locality using stack distances. 150–159.

[9] Siddhartha Chatterjee, Erin Parker, Philip J Hanlon, and Alvin R Lebeck. 2001. Exact analysis of the cache behavior of nested loops. *ACM SIGPLAN Notices* 36, 5 (2001), 286–297.

[10] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality analysis through static parallel sampling. *ACM SIGPLAN Notices* 53, 4 (2018), 557–570.

[11] Allen Cypher. 1995. Eager: Programming repetitive tasks by example. In *Readings in human–computer interaction*. Elsevier, 804–810.

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

[13] 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.

[14] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[15] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A fast analytical model of fully associative caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 816–829.

[16] 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)*. 351–364.

[17] Intel. [n.d.]. System Memory at a Fraction of the DRAM Cost. ([n. d.]). https://www.intel.com/content/dam/www/public/us/en/documents/brief/intel-ssd-software-defined-memory-with-vm.pdf

[18] Ken Kennedy and John R Allen. 2001. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc.

[19] 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.

[20] Tessa A. Lau and Daniel S. Weld. 1999. Programming by Demonstration: An Inductive Learning Formulation. In *Proceedings of the 4th International Conference on Intelligent User Interfaces* (Los Angeles, California, USA) *(IUI '99)*. ACM, New York, NY, USA, 145–152. https://doi.org/10.1145/291080.291104

[21] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. 2019. Beating OPT with statistical clairvoyance and variable size caching. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 243–256.

[22] Henry Lieberman. 1993. Tinker: A programming by demonstration system for beginning programmers. *Watch what I do: programming by demonstration* 1 (1993), 49–64.

[23] Fangzhou Liu, Dong Chen, Wesley Smith, and Chen Ding. 2020. PLUM: static parallel program locality analysis under uniform multiplexing. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 437–438.

[24] 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.

[25] David Maulsby and Ian H Witten. 1997. Cima: an interactive concept learning system for end-user applications. *Applied Artificial Intelligence* 11, 7-8 (1997), 653–671.

[26] Tom M Mitchell. 1982. Generalization as search. *Artificial intelligence* 18, 2 (1982), 203–226.

[27] Andreas Moshovos and Gurindar S Sohi. 1999. Read-after-read memory dependence prediction. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 177–185.

[28] Louis-Noël Pouchet. [n.d.]. PolyBench/C 4.2.1. http://polybench.sourceforge.net.

[29] Ian Prechtl, Ben Reber, Chen Ding, Dorin Patru, and Dong Chen. 2020. Clam: Compiler lease of cache memory. In *The International Symposium on Memory Systems*. 281–296.

[30] Gabriel Rodríguez, José M Andión, Mahmut T Kandemir, and Juan Touriño. 2016. Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 139–149.

[31] Gabriel Rodriguez, Mahmut T Kandemir, and Juan Tourino. 2018. Affine modeling of program traces. *IEEE Trans. Comput.* 68, 2 (2018), 294–300.

[32] David Canfield Smith. 1976. *PYGMALION: A Creative Programming Environment*. Technical Report. http://worrydream.com/refs/Smith%20-%20Pygmalion.pdf

[33] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5 (2013), 475–495.

[34] Phillip D. Summers. 1976. A Methodology for LISP Program Construction from Examples. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages* (Atlanta, Georgia) *(POPL '76)*. ACM, New York, NY, USA, 68–76. https://doi.org/10.1145/800168.811541

[35] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2017. Ascertaining uncertainty for efficient exact cache analysis. In *International Conference on Computer Aided Verification*. Springer, 22–40.

[36] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and exact analysis for LRU caches. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[37] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

[38] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. 30–44.

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

[40] 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 (TACO)* 14, 4 (2017), 1–26.

[41] Liang Yuan, Chen Ding, Wesley Smith, Peter Denning, and Yunquan Zhang. 2019. A Relational Theory of Locality. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 3 (2019), 1–26.