Synthesizing Symbolic Reuse Intervals for Loop Nests

ANONYMOUS AUTHOR(S)

As memory systems become more complex, there is an increasing demand for efficient and precise descriptions of data movement. However, past static techniques that rely on complex mathematical models are often limited to linear expressions, and approximations are required when solving for performance numbers.

In this paper, we tackle the locality analysis problem by turning it into a program synthesis problem using input-output examples. We suggest three specifications to describe symbolic representations for reuses and their input-output example structures. We use unification search with an elimination-free DSL for synthesis and introduce reuse-type inferred and code-structure inferred biases to guide the search. We present synthesized symbolic reuse intervals for 30 programs in PolyBench and compare predicted miss ratio curves with tracing.

Additional Key Words and Phrases: Locality, Program Synthesis, Static Analysis

1 INTRODUCTION

Modern applications require fast access to a large amount of data. To meet this requirement, heterogeneous architectures have been developed that augment traditional DRAM with emerging memory devices or new interconnection techniques such as storage-class memory, high-bandwidth memory (HBM), and disaggregated memory. These architectures provide capacity, low access latency, and high bandwidth memory subsystems. As a result, applications increasingly rely on locality analysis to better exploit the heterogeneous memory subsystem via loop transformation [Lam and Wolf 2004; Pouchet et al. 2011], data placement [Ahmadi et al. 2022; Chen et al. 2014], and runtime cache management [Avissar et al. 2001; Ye et al. 2017a].

Analytical cache modeling [Bao et al. 2017; Gysi et al. 2019; Khan et al. 2021; Morelli and Reineke 2022; Shah et al. 2022] is a promising technique that allows users to calculate the cache miss rate without simulating the cache behavior. This technique has received much recent attention and is a promising solution. Instead of simulating the cache behavior, analytical cache modeling describes a program's memory accesses using integer sets and relations of program variables [Bao et al. 2017]. When a program input is specified, it derives the cache miss rate by replacing the variables with concrete values.

Existing work has made significant advances in accelerating analytical cache modeling [Shah et al. 2022] or building models for hierarchical caches [Bao et al. 2017] or other cache architectures [Morelli and Reineke 2022]. However, the prominent uses of those solutions are still bound to affine programs such as stencil programs. That is because those solutions rely on polyhedral analysis to translate a Static Control Parts (SCoP) of a program into symbolic representations. A prior study has shown that only 275 (14.8%) out of 1,862 code regions in SPEC2000 are SCoP, which implies that the majority (85.2%) of code regions are presumably beyond the capability of existing analytical cache modeling techniques.

This paper presents the first synthesis approach for locality analysis that targets generic loop-based codes, i.e., has the potential to go beyond SCoP. Unlike previous approaches that rely on locality experts to understand the reuse relation of the references and infer the analytical cache model from the source code, the new approach is data-driven, i.e., it derives the analytical model from input and output examples extracted from a set of memory access traces of a program, which encode the reuse relation needed for modeling.

Specifically, the approach extracts a set of input-output *examples* by tracing a `loopCode` with different values for program parameters. The examples encode the detailed reuse relations for the

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

loop. With the examples, the synthesizer searches for a symbolic representation that satisfies all input-output examples. The generated symbolic representations with symbols from the `loopCode` are the analytical model that describes locality. The process is formalized as follows:

$$\texttt{Examples} \leftarrow \texttt{loopCode}$$

$$\texttt{Analytical Model} \leftarrow \textit{Synthesizer}(\texttt{DSL}, \texttt{Examples})$$

Realizing the synthesis approach faces three major challenges:

- Abstraction: how to extract a set of input and output examples from the target loop? The examples should be able to describe the locality character for the loop. At the same time, they should be able to generalize to an efficiently solvable analytical model.
- Efficiency: how to bound the complexity of the synthesis process to make the approach applicable?
- Precision: how to ensure the synthesized program correctly and precisely describes the locality, as the extracted examples are finite while the reuses in a loop nest can be unbounded?

The paper tackles the challenges with a set of novel techniques:

- It presents an abstraction for the input and output examples by formalizing the relationships between the locality character, reuse interval [Xiang et al. 2013] in this work, and input variables of the target loop in Section 3. From the relation, it proposed three specifications for the synthesized analytical model and the structure of input-output examples for them in Section 4.
- Regarding efficiency, it adopts syntax-guided unification search. To prune the search space, it proposes an elimination-free DSL with biases in Section 5.
- Regarding the precision, it empirically evaluates the quality of the synthesized analytical model with Polybench and a non-affine program, as shown in Section 6. The synthesized model can predict cache misses with comparable accuracy to tracing.

2 BACKGROUND

Table 1 highlights existing analytical cache models. The compiler researchers start to pay attention to this problem in order to guide loop transformations [Kennedy and McKinley 1992; Wolf and Lam 1991]. Optimizing compilers rely on data dependences between references to check the legality of a loop transformation. References that touch the same array element will have a data dependence. If a transformation does not revert any data dependence between array elements from references, the transformation is safe.

The dependence distance vector quantifies the iteration difference between two references. As an example, Fig. 1 shows the 5-point stencil kernel code and its dependence distances. Each iteration in Fig. 1a calculates the sum of a[i][j] and its four neighbors. The access to $a[i][j+1]$ at iteration i', j' will touch the same array element with the access to $a[i][j]$ at iteration $i', j'+1$. Their distance will be (0, 1), which indicates one iteration difference in j loop. Fig. 1b shows all distance vectors for all references to array a. Dependences identify the partial orders of all array elements in a loop. For the same reference, the shortest dependence distance vector (a.k.a. *reuse vector*, denoted as $\vec{sv}$ for the rest of this paper) indicates reuses, which guarantees there is no intervening access to the same array element.

The first five works in Table 1 are based on reuse vectors. They adopt different cost models to compute cache misses or cache line occupations from reuse vectors. [Wolf and Lam 1991] and [Kennedy and McKinley 1992] convert reuse vectors to the number of memory accesses per iteration with simple symbolic expressions. The symbols in the expression are loop bounds, strides, and cache line size. Substitution with concrete values will derive cache line usage for a loop in

Analytical Models	Requirement	Solver	Locality Representations
Uniformly Generated Sets [Wolf and Lam 1991]	Dependence Distance	No	Reuse Vector + Symbolic Cost Model
Loop Cost Functions [Kennedy and McKinley 1992]	Dependence Distance	No	Reuse Vector + Symbolic Cost Model
Cache Miss Equations [Ghosh et al. 1997]	Dependence Distance	Counting	Reuse Vector + Cost Model in LDE
Static Reuse Distance Histogram [Cascaval and Padua 2003]	Dependence Distance	Counting	Reuse Vector + Cost Model in Affine Sets
Static Reuse Interval Histogram [Chen et al. 2017]	Dependence Distance	No	Reuse Vector + Cost Model in HOTL
Reuse Distance Equation [Beyls and D'Hollander 2005]	SCoP	Counting	Affine Sets and Maps
PolyCache [Bao et al. 2017]	SCoP	Counting	Affine Sets and Maps
Haystack [Gysi et al. 2019]	SCoP	Counting	Affine Sets and Maps

Table 1. Overview of analytical cache models

(a) 5-point stencil kernel

```
// a, b are 2D 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

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)	-

Fig. 1. 5-point stencil kernel and its dependence distances (read-after-read dependence [Moshovos and Sohi 1999])

constant time. [Ghosh et al. 1997] and [Cascaval and Padua 2003] convert reuse vectors to a set of symbolic expressions in the form of Linear Diophantine Equations (LDE) or sets with symbolic ranges. They introduce loop induction variables to the equations to distinguish different iterations. Misses or distances are counted from their solution sets. [Chen et al. 2017] adopts a cost model that is commonly used in dynamic analysis on memory access traces.

Though we have methods to convert reuse vectors to derive cache performance, it is hard to derive the distance vectors when the loop structure is complex. Reuse vectors only capture the reuses between references whose index expressions differ by a constant. Even for indices i and $2 * i$, we can not summarize their distance as a constant vector for all iterations in a loop.

Optimizing cache performance for loops is not limited to automatic loop transformations which requires dependence distances. The cache models are also essentials for optimizations that do not

change the loop codes, such as cache hint generation [Beyls and D'Hollander 2005], cache partition [Brock et al. 2015], compiler leasing of a cache [Ding et al. 2022]. The last three approaches [Bao et al. 2017; Beyls and D'Hollander 2005; Gysi et al. 2019] in Table 1 focus on Static Control Parts (SCoP), whose loop bounds and array references are affine functions of loop iterators and program parameters [Grosser et al. 2011]. With extracted affine functions from the loop codes, they use linear constraints to formulate reuse relations. By solving the reuse formulation, they can derive a reuse vector for a specific iteration. Counting usually takes polynomial time for affine sets. Fig. 2 shows the reuse constraints from the Reuse Distance Equation for reference $a[i][j]$ and $a[i][j+1]$. To form a reuse, four constraints must be satisfied: (C1) both iteration vectors are within iteration space; (C2) the access to a[i][j] happens after the access a[i][j+1]; (C3) both accesses are to the same array element; (C4) there is no intervening access to the same array elements from other references.[1]

$$
\begin{aligned}
reuse_{a[i'][j'+1],a[i][j]} = \{(i,j,i',j') | & C1 \wedge C2 \wedge C3 \wedge C4\} \\
& C1 : 0 < i, j, i', j' < B + 1 \\
& C2 : (i', j') < (i, j) \\
& C3 : i = i' \wedge j = j' + 1 \\
& C4 : \texttt{No Intervening Accesses From Other References}\}
\end{aligned}
$$

Fig. 2. Constraints for reuses at $a[i][j]$ from $a[i'][j'+1]$

Constraints offer an iteration-based reuse relationship rather than a single reuse vector for all iterations. However, this approach has the disadvantage of obscuring the reuse relationship for each iteration. Counting constraints only reveals the number of solutions, not the solutions themselves. As a result, the reuse relationship is only visible as a summary of all reuse relationships between two references as the equation in Fig. 2 shows. This makes it difficult to understand the detailed reuse relationship for a specific access in a specific iteration. One could replace constraint C1 with a specific iteration to reveal the relationship. However, this would be costly as counting takes polynomial time.

Furthermore, having a compact representation can also hinder understanding of how cache performance changes with different program parameters. These parameters, such as loop bounds, are represented symbolically in the constraints. Only by solving the constraints for different bound values and obtaining concrete numbers, it is possible to see how cache performance changes. ***Is there an analytical model that is both symbolic and iteration-based without requiring counting?*** Here, we leverage the synthesizer to build symbolic models for each iteration automatically.

3 DESIGN

3.1 Reducing to A Synthesis Problem

Trace-based analysis can give us the precise information about the reuse relation for each access in a loop with specific program parameters. Given a memory access trace of a loop, the trace is scanned for reuses. There are two metrics: one is Reuse Distance (RD) [Bélády 1966] and the other is Reuse Interval (RI) [Xiang et al. 2013]. Reuse interval is *the number of memory accesses* between two consecutive access to the same data block. Simpler than reuse distance, reuse interval removes the requirement of counting distinct memory accesses in between. i.e., memory access trace "*abbba*"

[1]We hide this part for a better presentation.

will have a reuse interval with length 4 and a reuse distance 2 for block "a" [2]. For block "b", there are two reuses with both reuse interval and reuse distance equal to 1. To derive cache performance for this given trace, reuse intervals or reuse distances for all blocks are usually merged to form a histogram. i.e., reuse interval histogram is {1:2, 4:1} [3] and reuse distance histogram is {1:2, 2:1} for trace "*abbba*". With reuse distance or reuse interval histograms, the costs can be directly calculated by these metrics.

Reuse intervals can be calculated with reuse vectors and the number of accesses per loop iteration. Reuse vectors indicate the number of iteration difference between the reuse source and its sink. Combined with the number of accesses per loop iteration, the iteration difference converts to the number of intervening accesses in between. If we can locate a specific access in a trace with symbolic program constructs, we can build a symbolic RI histogram which satisfies our goal.

3.2 Identify Symbols for Synthesis

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 $\vec{\mathtt{B}}$.
(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 consider 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 $\vec{\mathtt{I}}_{\mathtt{snk}}$ locate 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 synthesis problem can be formulated as finding a program PROG written in a Domain-Specific Language (DSL) with the above factors as variables. 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.

[2] This calculation includes the first block "a". 4 accesses to 2 difference blocks are before the second "a"
[3] {1:2, 4:1} means 1 appeared twice and 4 appeared once.

3.3 System Framework

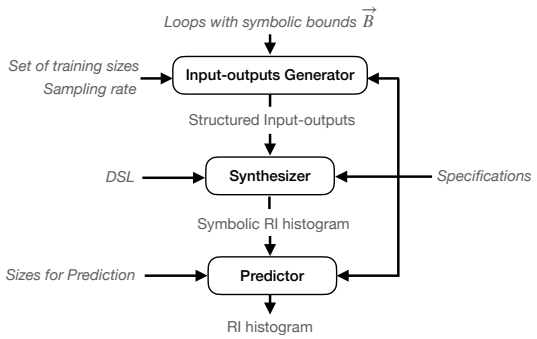

Fig. 3. System Framework

We propose a framework to resolve the synthesis problem. Figure 3 shows the core components:

- *Input-output example generator* takes the loop with symbolic bounds $\vec{B}$ and generates structured input-output examples that encode reuse relations. Different training sizes for $\vec{B}$ and different sampling rates for loop iterations can be provided.
- *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.
- *Predictor* takes in different $\vec{B}$ values and the synthesized analytical model. Then it directly derives their locality.

4 SPECIFICATIONS AND THEIR STRUCTURED INPUT-OUTPUT EXAMPLES

This section designs the specifications based on abstraction insights.

4.1 Specifications for Ruse Interval

We propose three specifications for a program PROG that describes a reuse interval RI. The specifications are differentiated in the information they capture. They follow the same format:

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

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 $\vec{B}$ 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 variables include all symbols from the source iteration vector and loop bound variables.

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

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.

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

Table 2. Input-output examples for Spec-src

input	$\vec{\mathtt{I}}_{\mathtt{src}}$	$\vec{\mathtt{B}}$	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 $\mathtt{Ref}_{\mathtt{src}} = c_0, \vec{\mathtt{I}}_{\mathtt{src}} = \vec{c}_1$				

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 using 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 $\vec{\mathtt{I}}_{\mathtt{snk}} - \vec{\mathtt{I}}_{\mathtt{src}}$.

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

4.2 Structured Input-Output Examples

To generate examples, we first profile the loop with different $\vec{\mathtt{B}}$ for reuses. The number of different training $\vec{\mathtt{B}}$ values determines the number of traces to collect. For example, if we provide three train sizes 4, 8, 16 for each element of $\vec{\mathtt{B}}$ and the length of $\vec{\mathtt{B}}$ is 2, we will have $3^2 = 9$ traces. The $\vec{\mathtt{B}}$ 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 the following run-time states of a loop: (1) the reference ID $\mathtt{Ref}_{\mathtt{src}}$ and iteration $\vec{\mathtt{I}}_{\mathtt{src}}$ of a reuse source; (2) the same information $\mathtt{Ref}_{\mathtt{snk}}$, $\vec{\mathtt{I}}_{\mathtt{snk}}$ for its reuse sink; (3) its reuse interval RI. Formally, a trace record format for a reuse is defined as:

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

Given the collection of traces, we extract and construct input-output examples for each program specification. Here we assume there are n different values for $\vec{\mathtt{B}}$.

Examples for Spec-src. With n traces, we extract the records whose $\mathtt{Ref}_{\mathtt{src}} = c_0$ and $\vec{\mathtt{I}}_{\mathtt{src}} = \vec{c}_1$ to form the examples for $\mathtt{RI}_{\mathtt{Ref}_{\mathtt{src}}=c_0, \vec{\mathtt{I}}_{\mathtt{src}}=\vec{c}_1}$ as Table 2 shows. The input symbols are $\vec{\mathtt{I}}_{\mathtt{src}}$, $\vec{\mathtt{B}}$, and the output symbol is RI. All records share the same source iteration, so all values for $\vec{\mathtt{I}}_{\mathtt{src}}$ in the input-output examples are the same. The values for $\vec{\mathtt{B}}$ will all be 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 $\mathtt{Ref}_{\mathtt{snk}}$. Table 3 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

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

input	$\vec{\mathtt{I}}_{\mathtt{src}}$	$\vec{\mathtt{B}}$	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 $\mathtt{Ref}_{\mathtt{src}} = c_0, \vec{\mathtt{I}}_{\mathtt{src}} = \vec{c}_1, \mathtt{Ref}_{\mathtt{snk}} = c_2$				

Table 4. Input-output examples for Spec-src-snk+

input	$\vec{\mathtt{I}}_{\mathtt{src}}$	$\vec{\mathtt{I}}_{\mathtt{snk}}$	$\vec{\mathtt{B}}$	output	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 $\mathtt{Ref}_{\mathtt{src}} = c_0, \vec{\mathtt{I}}_{\mathtt{src}} = \vec{c}_1, \mathtt{Ref}_{\mathtt{snk}} = c_2$					

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.

Note that it is possible that given a specific $\vec{\mathtt{B}}$ value, there is no reuse in the selected source iteration $\vec{c}_1$. In this case, we set RI to 0. 0 entries are essential in the input-output examples as, together with non-zero entries, it will indicate the conditions specified by $\vec{\mathtt{I}}$ and $\vec{\mathtt{B}}$, that 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 the 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.

4.3 Specification for RIH

With the above specifications for a specific RI, we can define the analytical model 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:

$$\mathtt{RIH}(\vec{\mathtt{B}}) = \{\mathtt{RI}_{\mathtt{Ref}_{\mathtt{src}},\vec{\mathtt{I}}_{\mathtt{src}}}(\vec{\mathtt{B}}) | \mathtt{Ref}_{\mathtt{src}} \in R, \vec{\mathtt{I}}_{\mathtt{src}} \in IS\}$$

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

$$\mathtt{RIH}(\vec{\mathtt{B}}) = \{\mathtt{RI}_{\mathtt{Ref}_{\mathtt{src}},\vec{\mathtt{I}}_{\mathtt{src}},\mathtt{Ref}_{\mathtt{snk}}}(\vec{\mathtt{B}}) | \mathtt{Ref}_{\mathtt{src}}, \mathtt{Ref}_{\mathtt{snk}} \in R, \vec{\mathtt{I}}_{\mathtt{src}} \in IS\}$$

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+.

Table 5. Input-output examples for iteration space of a loop with induction variable l

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

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 $\vec{B}$ values. To avoid enumerating, we need an approach to generalize the RIH from a subset of $\vec{B}$ values to all other possible $\vec{B}$ values. To achieve this goal, we first characterize how the shape of iteration space changes with $\vec{B}$ 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 4a shows, the lower bound f_L and upper bound f_U are functions of induction variables from outer k loops $\vec{I}_{0\sim k-1}$ and bounds $\vec{B}$ 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

Fig. 4. Iteration space rectangularization

To efficiently characterize the iteration space, we project iteration space to a rectangular shape, as Figure 4b shows. Loop bounds will now be the minimum 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:

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

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 5 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 the smaller iteration space in (c).

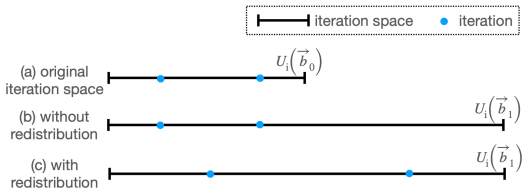

Fig. 5. Redistributing iterations for one-dimensional iteration space

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

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

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

5 SYNTHESIS FRAMEWORK

Once structure input-output examples are generated, we use a syntax-guided search with unification [Albarghouthi et al. 2013; Alur et al. 2015; Udupa et al. 2013].

5.1 Unification search with elimination-free DSL

Search with unification. Algorithm 1 demonstrates the search process. First, we try to find a program that satisfies all input-output examples with `BottomUpSearch` in Line 2. If a program is not found, we split the input-output examples into two sets: left L and right R in Line 4. We also 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 at Line 5 to find a boolean expression serving as 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 anyone program is not found, we do backtracking to Line 4 with different splitting strategies. If no program is found, the algorithm returns 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.

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
```

(3) **Half:** Sort the output values and split the examples into 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 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. [4] 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 ×, if-then-else ITE; Boolean operations negation ¬, and ∧, less than <. It is sufficient to cover all possible integer functions to describe

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

$$
\begin{aligned}
\texttt{IntExpr} &= \texttt{Num} \mid \texttt{Var} \mid \texttt{Plus} \mid \texttt{Times} \\
\texttt{Num}^{g'} &= \texttt{Num}^{g0} \times \texttt{Num}^{g1} \\
\texttt{Times}^{g'} &= (\texttt{Num} \mid \texttt{Var})^{g0,\texttt{lex0}} \times \texttt{TimesRight}^{g1,\texttt{lex1}} \\
\texttt{TimesRight}^{g'} &= \texttt{Var}^{g0,\texttt{lex0}} \times (\texttt{Var} \mid \texttt{TimesRight})^{g1,\texttt{lex1}} \\
\texttt{Plus}^{g'} &= (\texttt{Num} \mid \texttt{Var} \mid \texttt{Times})^{g0,\texttt{lex0}} + \texttt{PlusRight}^{g1,\texttt{lex1}} \\
\texttt{PlusRight}^{g'} &= (\texttt{Var} \mid \texttt{Times})^{g0,\texttt{lex0}} \\
&\quad + (\texttt{Var} \mid \texttt{Times} \mid \texttt{PlusRight})^{g1,\texttt{lex1}} \\
\\
\texttt{Lt}^{g'} &= \texttt{IntExpr}^{g0} < \texttt{IntExpr}^{g1} \\
\texttt{Not}^{g'} &= \neg \texttt{And}^{g0} \\
\texttt{And}^{g'} &= \texttt{Lt}^{g0,\texttt{lex0}} \ \wedge \ (\texttt{And} \mid \texttt{Not} \mid \texttt{Lt})^{g1,\texttt{lex1}}
\end{aligned}
$$

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

Fig. 6. Elimination-free DSL

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.

Figure 6 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 the 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. [5]

5.2 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 $\vec{\mathtt{B}}$s do not change. The other is *scaling* reuse interval, that the non-zero RI values for different $\vec{\mathtt{B}}$s scale with $\vec{\mathtt{B}}$ or $\vec{\mathtt{I}}_{\mathtt{src}}$.

For constant RI, the preferred program for it is a Num. For scaling RI, the preferred program should contain $\vec{\mathtt{B}}$ or $\vec{\mathtt{I}}_{\mathtt{src}}$ for Spec-src, Spec-src-snk, and $\vec{\mathtt{I}}_{\mathtt{snk}}$ 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:

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

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

If a RI scales with $\vec{\mathtt{I}}_{\mathtt{src}}$, the chance to successfully summarize it in $\vec{\mathtt{B}}$ is small as in the examples, the values for $\vec{\mathtt{I}}_{\mathtt{src}}$ are the same for all records but $\vec{\mathtt{B}}$ values differ from each other. For $\vec{\mathtt{B}}$ and $\vec{\mathtt{I}}_{\mathtt{src}}$, if the coefficients of indices of source and sink references' $\vec{\mathtt{I}}_{\mathtt{src}}$ are different or loop bounds are functions of $\vec{\mathtt{I}}_{\mathtt{src}}$s. The RI is more likely to scale with $\vec{\mathtt{I}}_{\mathtt{src}}$, otherwise we prefer $\vec{\mathtt{B}}$. $\vec{\mathtt{I}}_{\mathtt{snk}}$ only appears in programs under Spec-src-snk+. As $\vec{\mathtt{I}}_{\mathtt{snk}} - \vec{\mathtt{I}}_{\mathtt{src}}$ 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 $\vec{\mathtt{B}}$. 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 overlap with this path of will be irrelevant. We can safely ignore the loops and their bound symbols before the source reference and after 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

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

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.

5.3 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.

6 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 [Pouchet [n.d.]]. It contains 30 numerical kernels extracted from linear algebra, image processing, physics simulation, dynamic programming, and statistics applications. [6]

6.1 Case study with 5-point stencil

We generate the input-output examples for stencil in Figure 1a 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 are 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 $\vec{\mathtt{I}}_{\mathtt{src}}$ and $\vec{\mathtt{I}}_{\mathtt{snk}}$. 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 are 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 its 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 6 shows the number of programs constructed with a baseline `DSL` and its elimination-free form in the first

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

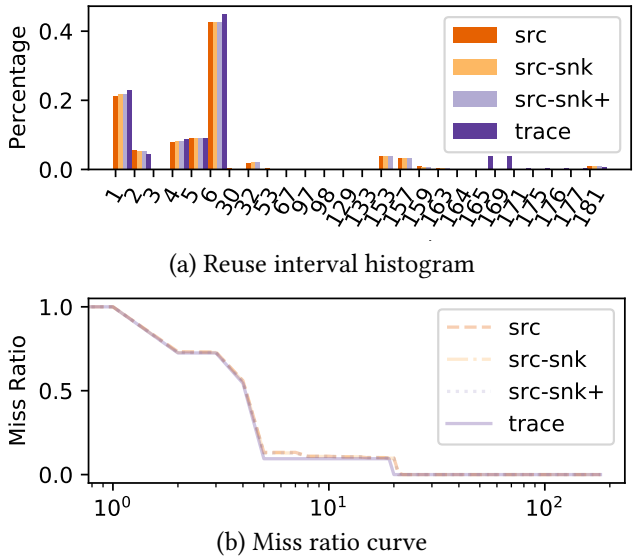

Fig. 7. Prediction with B = 32 by symbolic RIH specified by Spec-src, Spec-src-snk and Spec-src-snk+

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 6. 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

Predicting with RIH. By summarizing synthesized RIs, we get symbolic RIHs for all three specifications. Figure 7a shows the predicted reuse interval histograms from all three RIHs by assigning B to 32 and the trace. The x-axis shows the values of each reuse interval. All RIHs 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 7b shows the miss ratio curves derived from each histogram by average eviction time [Hu et al. 2016] for LRU fully associative cache. [7] All three miss ratio curves almost overlap with traced miss ratio curves.

6.2 Examples for PolyBench

We choose five train sizes, 4, 6, 8, 12, 20, for each bound in all benchmarks. Table 7 shows the size of generated structured input-output examples for all benchmarks.

[7] The miss ratios can be derived for multi-level set-associated cache [Ye et al. 2017b] and for parallelized loops [Liu et al. 2020] with RI distributions.

Table 7. 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
$\texttt{len}(\vec{B})$	4	5	2	2	2	1	2	2	2	3
#Shape	138	198	656	100	100	188	318	180	200	196
$\#\texttt{RI}_{spec1}$	555	809	1364	129	128	90	418	258	304	1199
$\#\texttt{RI}_{spec2/3}$	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
$\texttt{len}(\vec{B})$	4	5	2	2	2	1	2	2	2	3
#Shape	118	240	150	70	170	110	182	704	128	288
$\#\texttt{RI}_{spec1}$	55	214	383	285	224	137	323	4088	112	612
$\#\texttt{RI}_{spec2/3}$	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
$\texttt{len}(\vec{B})$	4	5	2	2	2	1	2	2	2	3
#Shape	246	324	76	228	306	132	164	80	76	86
$\#\texttt{RI}_{spec1}$	96	114	120	96	510	239	213	286	41	147
$\#\texttt{RI}_{spec2/3}$	89	104	120	110	517	238	245	288	38	155
$\#\vec{I}_{snk}$	258	248	240	220	1551	678	721	836	66	436

The number of symbolic bounds $\texttt{len}(\vec{B})$ ranges from 1 to 5. $\texttt{len}(\vec{B})$ 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 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 $\#\texttt{RI}_{spec3}$ by a factor of its vector length.

6.3 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. [8] Note that symbolic RI is only needed to be synthesized once for all future predictions. The synthesizing overhead can be amortized.

Figure 8 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.

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

[8] 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.

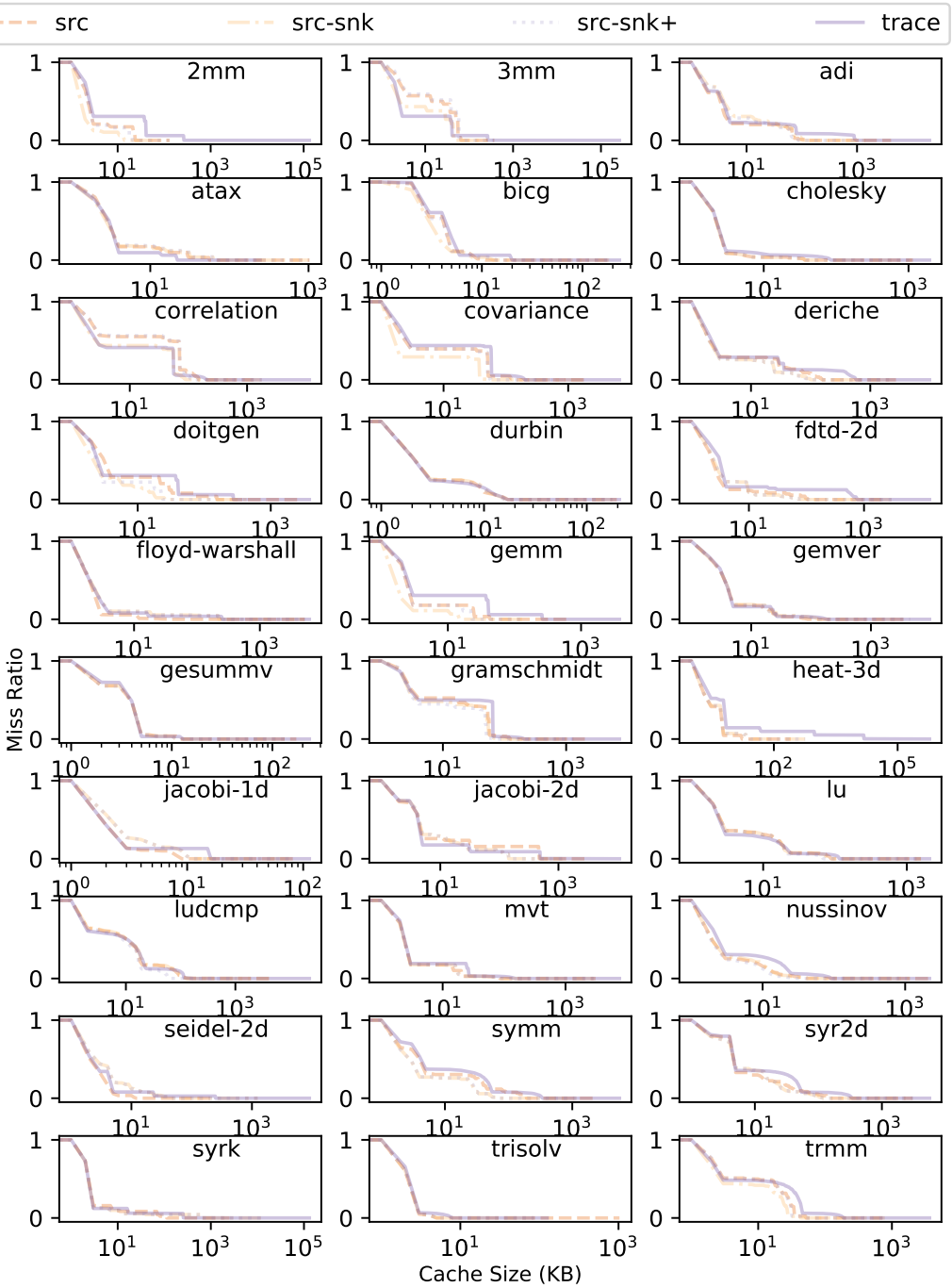

Fig. 8. Miss ratio curves predicted by synthesized reuse intervals

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 `RI`s for the examples/iterations with different search times or search biases. The newly searched programs can update a subset of the programs in the `RIH` without side effects.

6.4 Beyond SCoP

This technique can potentially handle non-SCoP loops, more specifically in the aspect of the following two cases:

Non-affine expressions for conditions and array indices. As Figure 9 shows, the bit reversal loop contains complex index calculations for j. The synthesizer can still capture its reuse pattern by iteration-based examples, as Figure 10 shows. As we adopt an iteration-based specification, the non-linearity for reuse intervals is already encoded in the examples.

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

Fig. 9. FFT bit reversal loop

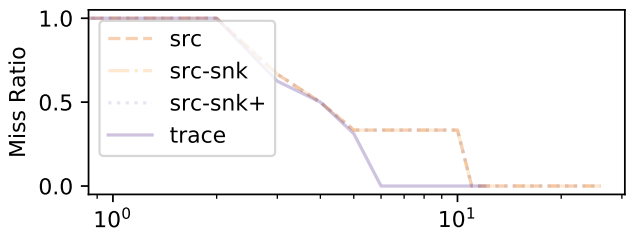

Fig. 10. Miss ratio cures for FFT bit reversal loop

Data-dependent control flow and indices. This pattern is common for graph algorithms and sparse matrices. When dependence happens on a single fixed array element, we can treat it as a variable to generate a program that considers its value. Such as $if(A[0] < 0)$ or $A[B[0]]$, we can generate $\mathsf{PROG}(v0)$ or $\mathsf{PROG}(v1)$, where $v0 = A[0], v1 = B[0]$.

For complex cases with dependence on multiple date elements, we need to figure out a function g, that can describe the relationship between the data values and their indices. Such as $if(A[i] < 0)$ or $A[B[i]]$, we can generate the same program $\mathsf{PROG}(v0)$ or $\mathsf{PROG}(v1)$. But with $v0 = g_A(i)$ or $v1 = g_b[i]$.

7 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 research has been exploring it, and the methods are becoming more and more sophisticated, from working set of loops [Ghosh et al. 1997; Kennedy and McKinley 1992; Wolf and Lam 1991] to reuse distance [Beyls and D'Hollander 2005; Cascaval and Padua 2003] and reuse interval [Chen et al. 2018] of reuses:

Estimating the working set: Wolf and Lam used reuse vectors, which are derived from dependence distance, to calculate the number of memory accesses for the innermost L loops [Wolf and Lam 1991]. Kennedy and McKinley proposed loop cost functions to calculate the number of cache lines accessed by the references in the innermost loop [Kennedy and McKinley 1992]. Ferdinand et al. proposed an abstract interpretation approach to track the working set in a set-associated cache [Alt et al. 1996]. Touzeau et al. further improved it by introducing new abstractions [Touzeau et al. 2017, 2019]. Ghosh et al. proposed cache miss equations to calculate misses from reuse vectors [Wolf and Lam 1991] for specific cache sizes, which is implemented in SUIF compiler framework [Ghosh et al. 1997]. Chatterjee et al. used Presburger formulas to express misses instead of using reuse vectors [Chatterjee et al. 2001]. Bao et al. proposed an integer-set-based model to calculate misses of polyhedral programs for set-associative cache [Bao et al. 2017].

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 [Cascaval and Padua 2003]. Beyls and D'Hollander proposed the reuse distance equation based on the polyhedral model to derive a reuse distance histogram [Beyls and D'Hollander 2005]. 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 [Gysi et al. 2019]. Chen et al. generate specialized loops to sample reuse intervals to construct a reuse interval histogram [Chen et al. 2018]. Instead of using mathematically models [Beyls and D'Hollander 2005; Cascaval and Padua 2003; Gysi et al. 2019], or sampler programs [Chen et al. 2018] to represent locality, our work generates iteration-based symbolic reuse intervals for the 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 [Smith 1976], Tinker [Lieberman 1993], Eager [Cypher 1995], Cima [Maulsby and Witten 1997]. PBE can also be encoded as an inductive learning process by enumerative search [Lau and Weld 1999]. It is often encoded with version space which learns a compact representation by provided general-to-specific relation [Mitchell 1982] or first-order logic to find a set of rules to describe the input-output relation. Such as THESYS learns LISP looping structures [Summers 1976], and FlashFill learns string processing programs [Gulwani 2011]. 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 [Rodríguez et al. 2016;

Rodriguez et al. 2018]. Instead of reconstructing the original code, our work searches for the locality representation of code.

8 CONCLUSION AND FUTURE WORK

This paper designs and implements the first input-output-examples-based synthesis system for locality analysis. It provides iteration-based symbolic reuse interval histograms, which allows characterizing the locality in fine-granted. 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. Getting locality information for a more complex program often requires trace analysis, such as programs containing alias, parallelism, or irregular accesses (sparse matrices). The proposed synthesis framework has the potential to consider all above challenges. It requires additional efforts to encode characters of input data or thread schedulers to the input-output examples.

Verification loop. Note that the current implementation generate symbolic representation with a fixed set of values for program parameters. This fixed set may not be sufficient to generate a precise symbolic RI for all iterations. We can introduce a verify loop into this framework by using a new program parameter that do not appear in the training set. As we are generating iteration-based symbolic RI, the iteration that fails to pass the verification can loop back to strengthen its input-output examples with the counter example. We can incrementally add new sizes for training to enlarge the examples until the predicted errors are tolerable.

Prescriptive cache management. The cache management in the past is reactive. Recent work shows the benefit of a prescriptive cache [Li et al. 2019; Prechtl et al. 2020]. [Li et al. 2019] shows the prescriptive fine-grained software management could achieve optimal cache performance with profile reuse intervals. [Prechtl et al. 2020] shows a hardware design for prescriptive cache. Our work provides symbolic iteration-based reuse information, which is essential for the fine-granularity prescription.

Program analysis through synthesis. Static locality analysis requires humans to examine the code in the past, visualize its execution in mind and build mathematical models. This paper converts the static locality analysis problem into a synthesis problem. Similar to program sketching [Solar-Lezama 2013], which leaves the algorithmic details to the synthesizer, our tool offloads model details to the synthesizer. We may generalize this approach to other analyses.

REFERENCES

Ali Ahmadi, Majid Daliri, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2022. Efficient approximations for cache-conscious data placement. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 857–871.

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

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

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

Oren Avissar, Rajeev Barua, and Dave Stewart. 2001. Heterogeneous memory management for embedded systems. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. 34–43.

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.

László A Bélády. 1966. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal* 5, 2 (1966), 78–101.

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

Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal Cache Partition-Sharing. In *2015 44th International Conference on Parallel Processing*. 749–758. https://doi.org/10.1109/ICPP.2015.84

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

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.

Dong Chen, Fangzhou Liu, Chen Ding, and Chucheow Lim. 2017. Static Reuse Time Analysis Using Dependence Distance. In *The 30th International Workshop on Languages and Compilers for Parallel Computing*.

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

Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An extensible optimizer for portable data placement on GPU. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 88–100.

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

Chen Ding, Dong Chen, Fangzhou Liu, Benjamin Reber, and Wesley Smith. 2022. CARL: Compiler Assigned Reference Leasing. *ACM Transactions on Architecture and Code Optimization* 19, 1 (2022), 1–28.

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.

Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly - Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*.

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

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.

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.

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.

Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 163–181.

Monica S Lam and Michael E Wolf. 2004. A data locality optimizing algorithm. *ACM SIGPLAN Notices* 39, 4 (2004), 442–459.

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

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.

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

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.

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.

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

Canberk Morelli and Jan Reineke. 2022. Warping cache simulation of polyhedral programs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 316–331.

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.

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

Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, Jagannathan Ramanujam, Ponnuswamy Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: convexity, pruning and optimization. *ACM SIGPLAN Notices* 46, 1 (2011), 549–562.

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.

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.

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

Nilesh Rajendra Shah, Ashitabh Misra, Antoine Miné, Rakesh Venkat, and Ramakrishna Upadrasta. 2022. BullsEye: Scalable and Accurate Approximation Framework for Cache Miss Calculation. *ACM Transactions on Architecture and Code Optimization (TACO)* (2022).

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

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

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

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.

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.

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

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

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

Chencheng Ye, Chen Ding, and Hai Jin. 2017a. Memory equalizer for lateral management of heterogeneous memory. In *Proceedings of the International Symposium on Memory Systems*. 239–248.

Chencheng Ye, Chen Ding, Hao Luo, Jacob Brock, Dong Chen, and Hai Jin. 2017b. Cache exclusivity and sharing: Theory and optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 4 (2017), 1–26.