Affine modeling of program traces

Gabriel Rodríguez, Mahmut T. Kandemir, *Fellow, IEEE,* Juan Touriño, *Senior Member, IEEE*

Abstract—A formal, high-level representation of programs is typically needed for static and dynamic analyses performed by compilers. However, the source code of target applications is not always available in an analyzable form, e.g., to protect intellectual property. To reason on such applications it becomes necessary to build models from observations of its execution. This paper presents an algebraic approach which, taking as input the trace of memory addresses accessed by a single memory reference, synthesizes an affine loop with a single perfectly nested statement that generates the original trace. This approach is extended to support the synthesis of unions of affine loops, useful for minimally modeling traces generated by automatic transformations of polyhedral programs, such as tiling. The resulting system is capable of processing hundreds of gigabytes of trace data in minutes, minimally reconstructing 100% of the static control parts in PolyBench/C applications and 99.9% in the Pluto-tiled versions of these benchmarks.

Index Terms—Program modeling, optimizing compilers, polyhedral optimization, memory traces.

1 Introduction

Affine codes represent an important class of applications in many computing domains, such as supercomputing, embedded systems, or multimedia applications. For the most part, these codes execute large regular loops, where the control- and data-flow can be exactly represented using affine functions of the loop index variables and loop invariant constants. These regions, often called Static Control Parts (SCoP) [9], are usually modeled and optimized using polyhedral compilation approaches [4, 8].

Many static and dynamic optimization and verification techniques rely on the knowledge of the application code to work. Unfortunately, the source code is not always available to the optimizer. In embedded systems for example it is common to find intellectual property (IP) cores with well defined high level functionality, but whose internals are opaque to the system designer and programmer. Organizations will not provide executables for privacy reasons, requiring researchers and contractors to deal with execution traces instead. Even when source code is available, it may not be amenable to static analysis and optimization, as programmers may use complex data and control structures, including code obfuscation techniques, that mask the underlying application logic.

This paper presents an analytical approach for automatically reconstructing an affine reference from a trace of its memory accesses. The Trace Reconstruction Engine (TRE) explores a tree-like space, in which level k contains all possible loops with trip count equal to k, from a 1-level nest iterating from 0 to $(k-1)$, to a k-level nest with a single iteration per level. The system is based on the observation that, in affine references, access strides must be constructed as linear combinations of loop index variables. The basic approach explores the entire solution space in a brute force fashion. On top of it, an exploration engine based on the mathematical properties of affine loops guides the process to achieve efficient reconstruction. Since the engine will eventually traverse the entire space, this process is guaranteed to find the minimal canonical affine loop nest that generates the exact input memory trace, given enough time. The main contributions of this work are:

- A mathematical framework for the construction of an affine representation of a given memory trace (Sec. 3), without user intervention or access to source codes or application binaries. Although compressing traces using affine representations is not a novel idea and has been explored in previous works [6, 7, 15], our proposal distinctly focuses on a single reference at a time. The backtracking mechanisms included in the reconstruction algorithm enable the construction of compact representations of complex traces, which cannot be achieved using other approaches.
- Extensions for the construction of unions of affine iteration domains to model piecewise-affine traces (Sec. 3.1). These types of traces are generated by codes which feature multiple lower and/or upper bounds in a single loop, combined using `max()` and `min()` functions, respectively, and are typically viewed as the union of canonical iteration domains. The exploration space is enlarged with respect to the single-domain affine case.
- A detailed experimental evaluation of the proposed technique (Sec. 4). Our results show that the framework can be used to build compact representations of large, complex traces, in acceptable time.

The framework can be potentially applied to guide all sorts of static and dynamic analyses and optimizations in the absence of source and/or binary codes, or when working with codes that are not amenable to static analysis for any reason. Examples of applications are automatic code optimization, hardware and software prefetching, data placement for locality optimizations, dependence analysis for automatic parallelization, optimal design of embedded

- *G. Rodríguez (corresponding author) and J. Touriño, Department of Computer Engineering, Universidade da Coruña, Spain; email: grodriguez@udc.es*
- *M.T. Kandemir, Department of Computer Science and Engineering, Pennsylvania State University, USA*

memory systems for locality, or trace compression. These applications are discussed in depth in Sec. 5, along with the related work.

This article builds on an earlier work [22], which covered the reconstruction of single-domain affine traces only. The approach in the current paper handles the general class of affine programs using unions of affine domains. New experimentation has been conducted modeling piecewise-affine traces, and previous experiments have been revised.

2 Problem Formulation

In the general case, the memory trace of a program contains all the memory accesses issued by its entire execution, including multiple loop and non-loop sections. In this paper it is assumed that each entry in the trace is labeled using an identifier of the instruction issuing the access, e.g., its memory address as done by Intel's Pin Tool [17]. The address stream generated by each memory instruction is analyzed separately. A mechanism to detect and extract loop sections in the trace [16, 18] may be used if a single instruction may appear in different loop scopes.

A general affine statement can be written as:

$$\begin{array}{l}\texttt{DO } i_1 = max(\ldots, l_{1,x}(\overrightarrow{i}), \ldots),\ min(\ldots, u_{1,x}(\overrightarrow{i}), \ldots)\\ \quad\vdots\\ \quad\texttt{DO } i_D = max(\ldots, l_{D,x}(\overrightarrow{i}), \ldots),\ min(\ldots, u_{D,x}(\overrightarrow{i}), \ldots)\\ \quad\quad V[f_1(\overrightarrow{i})]\ldots[f_N(\overrightarrow{i})]\end{array}$$

where $\{l_{(j,:)}, u_{(j,:)}; 0 < j \leq D\}$ are affine functions with rational coefficients[1]; $\{f_j(i_1, \ldots, i_D), 0 < j \leq N\}$ is the set of affine functions that converts a given point in the iteration space of the loop to a point in the data space of V; and $\overrightarrow{i}^k = \{i_1^k, \ldots, i_D^k\}^T$ is a column vector which encodes the state of each iteration variable for the k^{th} execution of V. For simplicity, we denote $l_j^k = max(\ldots, l_{j,x}(\overrightarrow{i}^k), \ldots)$ and $u_j^k = min(\ldots, u_{j,x}(\overrightarrow{i}^k), \ldots)$. Iteration bounds are assumed to be inclusive. Since each f_j is affine, the access can be rewritten as:

$$V[f_1(\overrightarrow{i})]\ldots[f_N(\overrightarrow{i})] = V[c_0 + i_1c_1 + \ldots + i_Dc_D] \quad (1)$$

where V is the base address of the array, c_0 is a constant stride, and each $\{c_j, 0 < j \leq D\}$ is the *coefficient* of the loop index i_j, and must account for the dimensionality of the original array[2].

During the execution of the loop, the access to V will orderly issue the addresses corresponding to $V(\overrightarrow{i}^1)$, $V(\overrightarrow{i}^2)$, etc. These addresses will be registered in the trace file together with the instruction issuing them.

2.1 Geometrical Considerations

In the polyhedral approach, each iteration of the former loop is modeled as an integer point in the D-dimensional space. The set of all loop iterations is then the intersection of an affine lattice and an integer polyhedron, resulting in a $\mathcal{Z}$–polyhedron [11]. Each of the F faces of a polyhedron can be identified with a hyperplane which divides the D-dimensional Euclidean space in two, and thus the $\mathcal{Z}$–polyhedron can be seen as the intersection of F half-spaces. In the context of the polyhedral model, each of the F faces corresponds to a lower or upper bound of an iteration index of the loop nest. In the following we will refer to these as the lower/upper bounds hyperplanes.

Consider two consecutive accesses, $V(\overrightarrow{i}^k)$ and $V(\overrightarrow{i}^{k+1})$, and assume that the loop index values in $\overrightarrow{i}^k$ and the upper and lower bounds functions are known. The values in $\overrightarrow{i}^{k+1}$ can be readily calculated as follows:

1) Index i_j will be reset if itself, and all inner indices, have reached their respective iteration upper bounds. Geometrically, $i_j^{k+1} = l_j^{k+1}$ *iff* $\overrightarrow{i}^k$ lies on an edge formed by the union of the upper bounds hyperplanes of the iteration polyhedron for dimension j and inner dimensions $(j+1), \ldots, D$:
$$\left(\forall x, j \leq x \leq D, u_x^k = 0\right)$$
2) Index i_j will increase by 1 if it has not yet reached its iteration upper bound, but all inner indices have. Geometrically, $i_j^{k+1} = i_j^k + 1$ *iff* $\overrightarrow{i}^k$ lies on an edge formed by the union of the upper bounds hyperplanes of the iteration polyhedron for inner dimensions $(j+1), \ldots, D$, but not on the hyperplane which serves as the upper bounds for dimension j:
$$\left(\forall x, j < x \leq D, u_x^k = 0\right) \wedge \left(u_j^k > 0\right)$$
3) In any other case, there are inner indices which have not yet reached their upper bounds, and therefore $i_j^{k+1} = i_j^k$.

Definition 2.1. *A set of indices built complying with these conditions will be referred to as a set of sequential indices.*

Consequently, the instantaneous variation of loop index i_j between iterations k and $(k+1)$, $\delta_j^k = (i_j^{k+1} - i_j^k)$, can only take one of three possible values:

1) i_j is reset to $l_j^{k+1} \Rightarrow \delta_j^k = l_j^{k+1} - i_j^k$
2) i_j is increased by one $\Rightarrow \delta_j^k = 1$
3) i_j does not change $\Rightarrow \delta_j^k = 0$

Lemma 2.2. *The stride between two consecutive accesses $\sigma^k = V(\overrightarrow{i}^{k+1}) - V(\overrightarrow{i}^k)$ is a linear combination of the coefficients of the loop indices.*

Proof. Using Eq. (1), σ^k can be rewritten as:

$$\begin{array}{rcccccl}\sigma^k = & V + (c_0 + & c_1 i_1^{k+1} & + \ldots + & c_D i_D^{k+1}) & - & \\ & V + (c_0 + & c_1 i_1^k & + \ldots + & c_D i_D^k) & = & \\ = & & c_1\delta_1^k & + \ldots + & c_D\delta_D^k & = & \overrightarrow{c}\,\overrightarrow{\delta}^k\end{array}$$

□

The single-domain integer affine class of loops is sufficient to model all the memory references in the PolyBench/C benchmarks [20]. An efficient algorithm to minimally reconstruct this class of loops is given in Sec. 3.

1. We use $u(\overrightarrow{i})$ to simplify notation, even though we should formally write $u(i_1, \ldots, i_{j-1})$. Coefficients of out-of-scope indices are assumed to be 0. The same notation is applied to lower bounds.

2. For instance, an access $A[2 * i][j]$ to an array $A[N][M]$ can be rewritten as $A[(2 * M) * i + j]$, where $c_i = 2M$ accounts for both the constant multiplying i in the original access (2), and the size of the fastest changing dimension (M).

Section 3.1 details the necessary considerations to consider the general class of affine loops (i.e., using union of iteration domains). While piecewise-affine loops are scarcely used in hand-written codes, they often appear when automatically applying compiler optimizations such as polyhedral loop tiling [10, 24].

3 Loop Synthesis

The proposed synthesis method is essentially a guided exploration of the potential solution space, driven by the first-order differences of the addresses accessed by a given instruction, i.e., the access strides. Each node in this space represents a convex polyhedron which corresponds to a portion of the entire trace to be reconstructed. The possible paths forward from each node are all the convex polyhedra in which a single point has been added with respect to said node. A geometrical depiction of this concept is shown in Fig. 1, and a more general view is given in Fig. 2. Starting from the root, a trivial loop which generates the first two accesses in the trace, the TRE incorporates one access to the reconstructed loop in each step, until it finds a solution for the entire trace or determines that no affine loop is capable of generating the input memory trace. The algorithm builds the minimal loop capable of generating the observed access trace[3]. This section develops the algebraic tools that allow to efficiently traverse the solution space.

Let $\mathcal{A} = \{a_1, \ldots, a_P\} = \left\{V(\overrightarrow{\imath}^1), \ldots, V(\overrightarrow{\imath}^P)\right\}$ be the sequence of addresses generated by a single reference in a single loop scope, extracted from the execution trace. The reconstruction algorithm iteratively constructs a solution $\mathcal{S}_D^P = \{\overrightarrow{c}, \mathbf{U}, \overrightarrow{w}\}$, which generates $\mathcal{A}$ using D nested loops. The components of this solution are defined as follows:

- Vector $\overrightarrow{c} \in \mathbb{Z}^D$ of coefficients of loop indices.
- Matrix $\mathbf{U} \in \mathbb{Z}^{F \times D}$, and vector $\overrightarrow{w} \in \mathbb{Z}^F$, the upper bounds matrix and vector, respectively.

The iteration domain $\mathbf{I}$ is an integer polyhedron with F bounding hyperplanes containing the iteration vectors $\overrightarrow{\imath} \in \mathbb{Z}^D$ such that:

$$\mathbf{U}\overrightarrow{\imath} + \overrightarrow{w} \geq \overrightarrow{0}^T \tag{2}$$

where each row $U_{(j,:)}$ of the bounds matrix encodes the coefficients of the j^{th} bounds hyperplane, while w_j contains its independent term.

The access strides generated by a valid solution $\mathcal{S}_D^P$ must match the input access trace. Using Lemma 2.2 this can be expressed as:

$$\overrightarrow{c}\mathbf{I} = \mathcal{A} \Leftrightarrow \overrightarrow{c}(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^k) = \overrightarrow{c}\,\overrightarrow{\delta}^k = \sigma^k, \forall k \in [1, P)$$

The proposed synthesis method proceeds iteratively, constructing partial solutions for incrementally larger parts of $\mathcal{A}$. The first partial solution is built as follows:

$$\mathcal{S}_1^2 = \left\{\overrightarrow{c} = \left[\sigma^1\right], \mathbf{U} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \overrightarrow{w} = [0, 1]\right\} \tag{3}$$

3. For example, a 2-level loop with indices i and j might iterate sequentially over the elements in array $A[N][M]$ if the upper bounds are defined as $u_i = N$, $u_j = M$ and the access is $V[i * M + j]$. This can be rewritten as a 1-level loop with index i, using $u_i = N * M$ and access $V[i]$.

or, equivalently:

$$\begin{array}{c} \texttt{DO } i_1 = 0,\ 1 \\ a_1 + \sigma^1 i_1 \end{array}$$

Starting from this first partial solution the engine begins working, gradually increasing its size, until it reaches a solution for the entire trace. Upon processing access a_{k+1}, the algorithm first calculates the observed access stride, $\sigma^k = a_{k+1} - a_k$, and builds a diophantine linear equation system based on Lemma 2.2 to discover the potential indices $\overrightarrow{\imath}^{k+1}$ which generate an access stride that is equal to the observed one:

$$\overrightarrow{c}(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^k) = \sigma^k \Rightarrow (\overrightarrow{c}^T\overrightarrow{c})\,\overrightarrow{\delta}^k = \overrightarrow{c}^T\sigma^k \tag{4}$$

where $(\overrightarrow{c}^T\overrightarrow{c}) \in \mathbb{Z}^{D \times D}$ is the system matrix, and $\overrightarrow{\delta}^k \in \mathbb{Z}^D$ is the solution. There are two possible situations when solving this system:

1) The system has one or more integer solutions. In this case, for each solution $\overrightarrow{\delta}^k$, the new index $\overrightarrow{\imath}^{k+1} = \overrightarrow{\imath}^k + \overrightarrow{\delta}^k$, which must be sequential to $\overrightarrow{\imath}^k$, is calculated. $\mathbf{U}$, $\overrightarrow{w}$, and $\overrightarrow{c}$ remain unchanged. Each of these solutions must be explored independently.
2) The system has no solution generating a sequential index. In this case, it is always possible to incorporate the next element in the trace by increasing the dimensionality of the synthesized loop. Because of the added computational complexity associated to dimensionality increases, the system may decide to backtrack to a previously generated partial solution, as explained in Sec. 3.2.

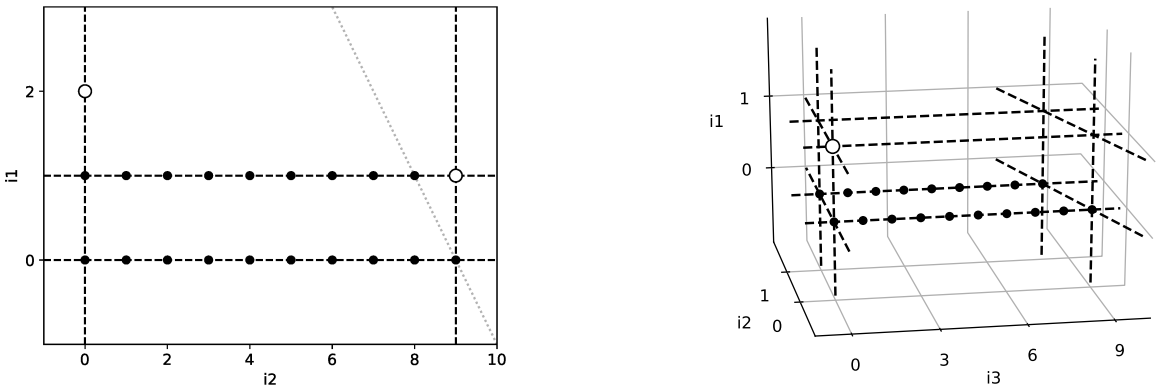
Although this diophantine system has infinite solutions in the general case, the actual number of valid solutions is limited by Def. 2.1. In fact, in order for the newly computed $\overrightarrow{\imath}^{k+1}$ to be sequential to $\overrightarrow{\imath}^k$, only D valid solutions exist, each of them of the form:

$$\left\{\left[\ \ldots \quad i_{j-1}^k \quad i_j^k + 1 \quad l_{j+1}^{k+1} \quad \ldots\ \right], 0 < j \leq D\right\} \tag{5}$$

This property allows a very efficient exploration of the solution space. However, the total number of generated alternatives is still large enough that traversing the solution space in a breadth-first fashion is not practical. The following guidance heuristic is incorporated: the system assumes the currently computed iteration polyhedron bounds to be correct, and explores the iteration index $\overrightarrow{\imath}^{k+1}$ generated by applying the rules in Sec. 2.1. If the generated index matches the next element in the trace, the exploration continues. Only if this fails will the engine generate the diophantine system and test all its possible valid solutions.

3.1 Computing Piecewise-Affine Iteration Bounds

When the guidance heuristic described in the previous section fails, the generated $\overrightarrow{\imath}^{k+1}$ will not be the one predicted by the current iteration bounds. $\mathbf{U}$ and $\overrightarrow{w}$ will need to be recomputed to ensure that the synthesized loop corresponds to the explored set of indices. This is done by recomputing the bounds hyperplanes which are not coherent with the newly generated iteration point iteratively, rotating the hyperplane in each step so that it contains the point furthest



(a) 2-d solutions (b) 3-d solutions

Fig. 1. Geometrical example of solution space. Assume that the reconstruction has reached a point represented by the iteration polyhedron $\mathbf{I}$ including the black dots in Fig. 1a. Its tentative bounding half-spaces, represented by dashed lines, are $(i1 \geq 0)$, $(i1 \leq 1)$, $(i2 \geq 0)$, and $(i2 \leq 9)$. There are 5 options for adding a point to $\mathbf{I}$. The first two ones are depicted as hollow dots in Fig. 1a, and correspond to adding points $(1, 9)$ or $(2, 0)$. Both will produce a potentially different stride in the access to V, depending on the access function f, which will be matched to the stride in the memory trace to assess its correctness. Note that, if $(2, 0)$ is selected, the upper bounds hyperplane for $i2$ will change to $(i2 \leq 9 - i1)$ (dotted line in the figure). The other three solutions correspond to dimensionality increases. One of them, depicted in Fig. 1b, corresponds to adding a new dimension to $\mathbf{I}$ in a way that it represents the outer loop of the nest (since point $(0, 1, 8)$ is followed by $(1, 0, 0)$). The access function f will be modified to include $i3$ matching the stride in the memory trace. Note that two additional solutions like this one exist, in which the new loop is added as the middle (the new point is $(1, 1, 0)$), or as the inner loop (the new point is $(1, 8, 1)$).

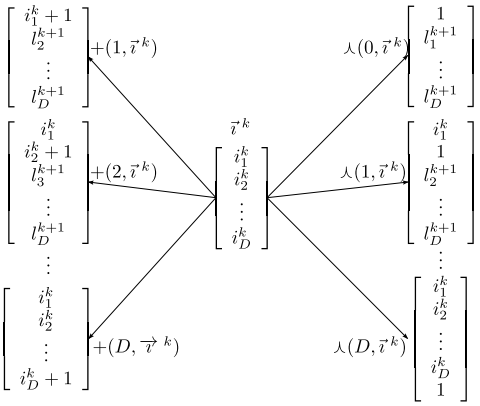


Fig. 2. Generic solution space. For each index $\overrightarrow{\imath}^k$, there are $(2D + 1)$ possible values for $\overrightarrow{\imath}^{k+1}$. The D alternatives on the left side are obtained using an operation $+(j, \overrightarrow{\imath})$ that increases index i_j by one, and resets all inner indices. The $(D+1)$ alternatives on the right are obtained by applying an operation $\lambda(j, \overrightarrow{\imath})$, which inserts a new loop at level $(j + 1)$. For instance, if $\overrightarrow{\imath}^k = [3, 5, 7]$ and lower bounds were 0, there are 7 alternatives for $\overrightarrow{\imath}^{k+1}$: $+(1, \overrightarrow{\imath}^k) = [\mathbf{4}, 0, 0]$, $+(2, \overrightarrow{\imath}^k) = [3, \mathbf{6}, 0]$, $+(3, \overrightarrow{\imath}^k) = [3, 5, \mathbf{8}]$, $\lambda(0, \overrightarrow{\imath}^k) = [\mathbf{1}, 0, 0, 0]$, $\lambda(1, \overrightarrow{\imath}^k) = [3, \mathbf{1}, 0, 0]$, $\lambda(2, \overrightarrow{\imath}^k) = [3, 5, \mathbf{1}, 0]$, and $\lambda(3, \overrightarrow{\imath}^k) = [3, 5, 7, \mathbf{1}]$.

away from it (in a way similar to Quickhull [3]). This process repeats until a valid solution is found; otherwise there is no convex polyhedron containing all required points. But it is still possible that a piecewise-affine solution exists.

Piecewise-affine loops are formed by the union of affine iteration domains (typically using `max()` and `min()` functions on loop bounds). Discovering new bounds is a relatively simple process: if rotating a bounds hyperplane fails to make the iteration polyhedron consistent with the generated iteration vectors, the engine will try to introduce a new hyperplane so that the problematic iteration point lies on its surface, and contains all other points in $\mathbf{I}$. For upper bounds hyperplanes this is a lightweight operation: as illustrated by Eq. (5), only D sequential indices exist under previously known lower bounds. However, discovering new lower bounds is a more complex problem. In this situation, the values for the indices which are reset on Eq. (5) are unknown, and must be discovered by solving an underdetermined equation system. Mathematically, this can be modeled by modifying the $+(j, \overrightarrow{\imath})$ operation in Fig. 2 so that it increases index i_j by one but, instead of resetting all inner indices to the currently known lower bounds l_j, resets them to unknown values $(i_{j+1}^{k+1}, \ldots, i_D^{k+1})$ such that $\overrightarrow{\imath}^{k+1}$ matches the observed stride:

$$\overrightarrow{c}(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^{k}) = \begin{bmatrix} c_j & \ldots & c_D \end{bmatrix} \begin{bmatrix} 1 \\ i_{j+1}^{k+1} - i_{j+1}^{k} \\ \vdots \\ 1_D^{k+1} - i_D^{k} \end{bmatrix} = \sigma^k$$

This system is underdetermined for any value $(j < D - 1)$. This implies that, in the general case, there exist infinite $\overrightarrow{\imath}^{k+1}$ candidates that provide the observed stride. In order to reduce the possibilities to a tractable set, two restrictions are introduced:

1) Only K unknowns in the set $(i_{j+1}^{k+1}, \ldots, i_D^{k+1})$ are allowed to reset to a value different from the ones predicted by the currently known lower bounds. In our experimental tests with PolyBench/C (see Sec. 4.2), $K = 3$ is enough to ensure optimal reconstruction of all references. Smaller values of K will result in less branching, but will potentially cause no solutions to be found within reasonable time or memory constraints for some traces.
2) Candidate vectors are restricted to those inside or adjacent to the iteration polyhedron projected by the current bounds (i.e., no point may exist in $\mathbb{Z}^D$ in between a candidate vector and the projected

polyhedron). This achieves optimal reconstruction of all references in PolyBench/C.

Once a valid $\overrightarrow{i}^{k+1}$ is determined, a new bounds hyperplane is computed iteratively as previously described. The process ensures that no points previously outside the iteration domain are now included in it, by successively adding as many new faces as necessary to preserve the original bounds.

3.2 Algorithm

Algorithm 1 presents the pseudocode of the Trace Reconstruction Engine (TRE). Esentially, the processing starts with an empty SCoP, and tries to enlarge it by sequentially adding points in the trace inside the `add_iter` call in line 14: in line 2 all the indices lexicographically following the most recent one are generated, while the loop in line 3 checks whether each of the generated indices explains the next value $\overrightarrow{i}^{k}$ in the trace. A list of candidate SCoPs is maintained and sorted by fitness heuristics. Whenever a solution cannot be found by building over the best ranked candidate, control will return to the `TRE` function. This function will retrieve the best ranked candidate in line 11, increase its dimensionality to incorporate one new point to the SCoP, and continue processing it. Line 4, which integrates the new iteration vector in the iteration domain, includes the potential modification of loop bounds and could fail if a non-convex polyhedron is generated.

Algorithm 1: Pseudocode of the TRE

```
   Input: the access trace, A; an input SCoP S
   Output: a SCoP reproducing the accesses in A
1  Function add_iter(A, S)
2      Find L = {i^{k+1}} lexicographical succesors of i^k;
3      for i^{k+1} ∈ L such that i^{k+1} c^ = a^k do
4          S' = S ∪ i^{k+1};
5          S_list = S_list ∪ add_iter(A, S')
6      end
7  end
8  Function TRE(A)
9      S_list = {EMPTY_SCoP};
10     while True do
           // retrieve the best ranked SCoP
11         S = retrieve_best(S_list);
12         if (#D^S == len(A)) then return S;
           // add dimension to include a^k
13         S = increase_dimensionality(S, a^k);
14         add_iter(A, S);
15     end
16 end
```

The previous pseudocode is a high-level representation of the actual implementation of the TRE. In fact, instead of generating a single lexicographical successor as shown in line 2, the engine streamlines the analysis by generating a slice of values corresponding to a full iteration of the outer loop, assuming that the currently known bounds are correct. If the memory accesses generated by that slice match the observed trace, the entire slice is incorporated and the process continues. Otherwise, the granularity of the generated slice is lowered, and the process repeated. Eventually, the trace is processed at the single entry level if necessary. Once the problematic region is analyzed, the size of the generated slices becomes larger to increase performance.

When the guidance heuristic detailed in Sec. 3 works flawlessly, the algorithm finishes in time $O(P)$ (the number of points in the trace). If the heuristic were to fail systematically, the algorithm would finish in $O(D^P)$. This behavior would be very rare, and imply a complete lack of regularity in the trace. From the memory perspective, the current implementation of the TRE requires to load into memory: i) the entire trace to be reconstructed; ii) $\mathbf{U}$, $\overrightarrow{w}$, and $\overrightarrow{c}$ for each branch explored in the reconstruction tree; and iii) a matrix containing the subset of surface points in the iteration polyhedron of the branch being currently explored, as they are needed when recomputing iteration bounds. This is by far the largest of the structures manipulated during the synthesis process. Other intermediate structures such as equation systems never contain more than a few dozen elements, and they are not relevant from the total memory point of view.

4 Experimental Results

The proposed TRE algorithm has been implemented in Python and applied to the PolyBench/C 4.2.1 suite [20]. It includes 30 applications from domains such as linear algebra, stencil codes, and data mining. The reconstruction algorithm was run for one reference of each loop scope in the static control parts of these applications (enclosed within `scop` pragmas). The entire memory access trace for each reference was stored in memory before being processed. The "large" problem size was used, except for `floyd-warshall` ("medium" size), which generates traces one order of magnitude larger than the second largest benchmark, taking up more than the available RAM. The characteristics of the traces for each benchmark are broken down in Table 1. Each execution was performed on an Intel Core i7 8700K Coffee Lake 3.70 GHz, with 64 GB of RAM.

4.1 Single-Domain Traces

Figure 3 shows aggregated trace sizes and processing times for each application. These largely depend on the number of reconstructed loops, as well as on the iteration pattern. For instance, the most efficient reconstruction is achieved for one of the references in `deriche`, an edge detection filter accessing arrays with a constant, single stride. The resulting trace is therefore trivial to recognize and is processed at 8.5 billion accesses per second. Disregarding single stride references, the most efficient reconstruction is achieved for one of the `fdtd-2d` references, a 2-d finite-difference time-domain kernel. This originally 3-d loop is reconstructed as a 2-d loop (the two inner ones are coalesced into a single one) in which the outer loop iterates only once per each 1.2 million iterations of the inner one. As a result, the reconstruction process can be largely streamlined: the trace contains blocks of 1.2 million elements separated by the same stride. Its 600 million accesses are sequentially processed in 20 milliseconds. Note that these numbers are referring to individual references contained in each application, while the figures show the aggregated values.

On the opposite end, one of `doitgen`'s references, emitting 3.4 million addresses, is the one processed at the slowest rate. It features a 2-level loop nest where the largest block

TABLE 1
Characteristics of the PolyBench/C benchmarks. The TRE was run for one sample reference of each loop nesting level in each benchmark SCoP. The total number of these sampled references for each benchmark is labeled as `#Scopes` in the table.

	Original (Sec. 4.1)			**Tiled** (Sec. 4.2)		
Benchmark	*#Scopes*	*#Accesses* ($\times 10^6$)	*Max. depth*	*#Scopes*	*#Accesses* ($\times 10^6$)	*Max. depth*
2mm	4	1657.58	2	6	1656.96	6
3mm	6	2702.59	3	6	2701.71	6
adi	6	1993.01	3	12	996.00	5
atax	4	7.98	1	4	7.984	4
bicg	3	3.99	1	4	7.984	4
cholesky	4	1335.33	3	10	1363.90	6
correlation	8	1012.92	3	9	1013.64	6
covariance	5	1012.92	3	7	1013.64	6
deriche	6	53.08	2	3	26.54	4
doitgen	3	544.32	3	3	544.32	4
durbin	4	6.00	2	4	6.00	2
fdtd-2d	4	1798.40	3	7	598.93	6
floyd-warshall	1	125.00	2	1	125.00	5
gemm	2	1321.10	3	2	1321.10	6
gemver	4	12.00	2	3	8.00	4
gesummv	2	1.69	1	5	3.38	4
gramschmidt	6	1441.92	3	5	1441.92	5
heat-3d	2	1643.03	4	86	842.23	8
jacobi-1d	2	2.00	2	6	1.00	4
jacobi-2d	2	1684.80	3	23	843.70	6
lu	3	2666.67	3	13	2666.67	6
ludcmp	8	2672.67	3	15	2676.66	3
mvt	2	8.00	2	1	4.00	4
nussinov	5	2610.41	3	6	2604.17	5
seidel-2d	1	1996.00	3	1	1996.00	6
symm	2	600.60	3	5	600.60	3
syr2k	2	721.32	3	2	721.32	6
syrk	2	721.32	3	2	721.32	6
trisolv	2	2.00	2	5	2.00	4
trmm	2	600.60	3	2	600.60	6

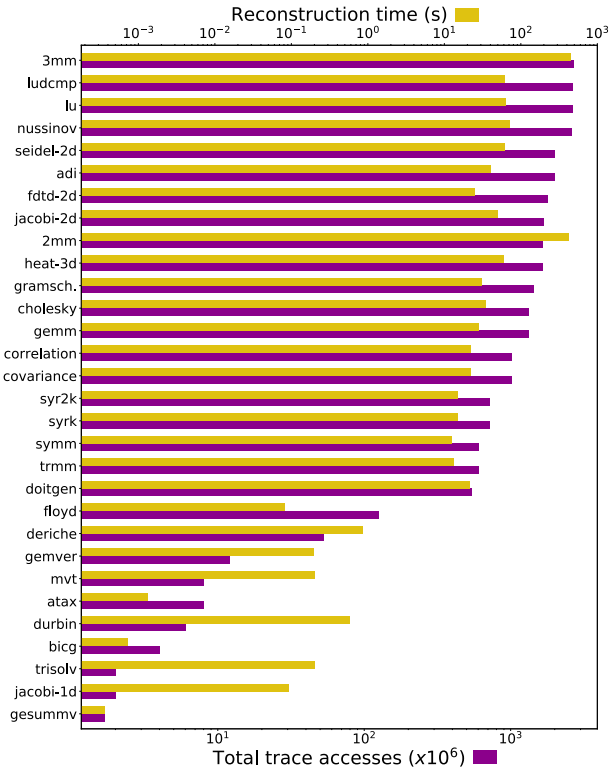


Fig. 3. Reconstruction times (upper axis) and trace sizes (lower axis) for PolyBench/C benchmarks, ordered by trace size. Axes are logarithmic.

of single-strided accesses contains only 160 elements. As such, the number of outer loop iterations, and consequently generated slices, is much larger. While in the slowest non-single-strided case the engine is capable of processing 2 million accesses per second, in the fastest one this figure goes up to 3 billion accesses per second (1500x faster). The entire aggregated input is processed at a rate of 20 million accesses per second.

A straightforward use of affine modeling is memory trace compression. We compared raw sizes, sizes using NumPy's `NPZ` (which uses `gzip`), and the sizes required to store $\mathbf{U}$, $\overrightarrow{w}$, and $\overrightarrow{c}$, which are enough to reconstruct the entire trace. The entire experimental set, which is 230 GB in size and can be compressed into 14.5 GB using `NPZ`, takes up 14 kB when compressed using the affine loop bounds reconstructed by the TRE. This represents a 17.2×10^6 and 1.1×10^6 compression factor with respect to the raw data and `NPZ`, respectively.

4.2 Piecewise-Affine Traces

Figure 4 details the reconstruction performance of tiled PolyBench/C 4.2.1 benchmarks. Pluto 0.11.4 was used to tile the static control parts in PolyBench/C using the `--tile` parameter. No parallelization or vectorization was performed. The figure clearly shows how the performance of recognizing the traces of tiled codes has decreased, in general, with respect to the original, untiled ones. The best performance is again obtained for a single-strided trace from `deriche`, recognized at a speed of 8.5 billion accesses per second. If we focus on non-single strided accesses, the

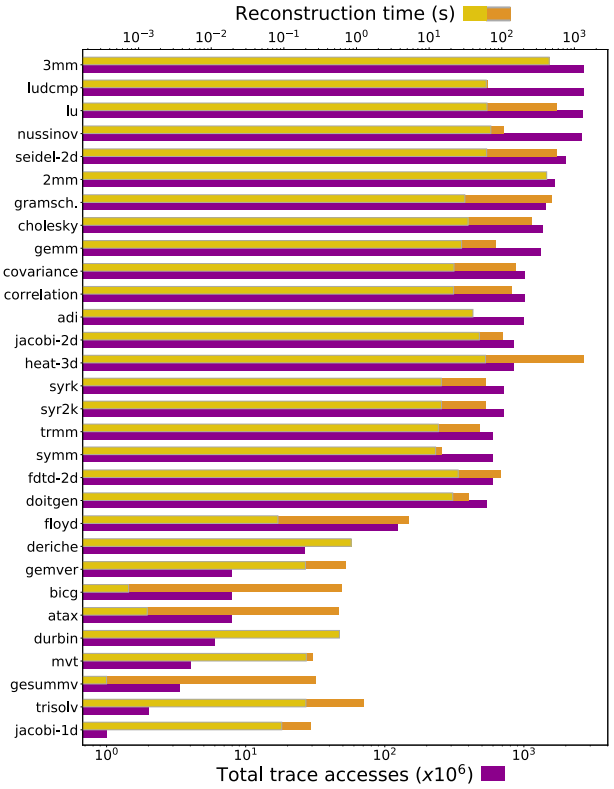


Fig. 4. Reconstruction times (upper axis) and trace sizes (lower axis) for the tiled PolyBench/C benchmarks, ordered by trace size. Axes are logarithmic. The left bar in the time axis represents the reconstruction time for the original, untiled code and is included for reference.

fastest reconstruction is achieved for one of the references in `ludcmp`, an LU decomposition followed by forward substitution, embedded in a triangular 3-d loop. The trace is reconstructed at 43 million accesses per second. On the opposite situation, one of the references of `heat-3d`, a stencil code solving the 3-d heat equation, is reconstructed at a rate of only 7000 accesses per second. While this is a small subtrace, reconstructed in under 3 minutes, it exemplifies one of the added difficulties of processing high dimensionality traces. This is a 6-dimensional loop. Not only the complexity of the equation systems is increased, but also the number of potentially valid paths. The entire aggregated input is processed at a rate of 6.1 million accesses per second, 3 times slower than in the single-domain case. Furthermore, 4 small references in the `heat-3d` benchmark (out of the total 258 analyzed, representing 0.1% of the total experimental data volume) cannot be reconstructed by the current implementation of the TRE, as memory is exhausted before finding a solution.

As for trace compression, we again compare raw sizes, sizes using `NPZ` compression, and the sizes required to store $\mathbf{U}$, $\overrightarrow{w}$, and $\overrightarrow{c}$. The experimental set now takes up 202 GB; 8.1 GB using `NPZ`; and 1.22 MB when reconstructed using the TRE. This represents a 1.7×10^5 and 6.8×10^3 compression factor with respect to raw data and `NPZ`, respectively.

5 Related Work and Applications

Several works have explored the representation of traces as loops to achieve benefits such as compression or program optimization. Clauss et al. [6, 7] characterized program behavior using polynomial piecewise periodic and linear interpolations separated into adjacent program phases to reduce function complexity. Ketterlin and Clauss [15] proposed a method for trace prediction and compression based on representing memory traces as sequences of nested loops with affine bounds and subscripts. From an input trace containing multiple references, they synthesize a program that generates the same trace when executed. Interestingly, although the objectives are very similar to our work, their approach is very different. As opposed to the single-reference approach followed by TRE, this work models full traces using imperfectly nested loops, without pre- or post-processing steps. A stack of terms (trace entries) is used, searching for triplets that can be rewritten as a loop. Non-minimal solutions may be found due to the greedy approach to merging triplets, or if some algorithmic parameters (e.g., the window size) are not large enough to detect regularity. We applied the approach in [15] to the same input traces as provided to the TRE in Sec. 4, with a window size of 100 terms. Single-domain traces are minimally reconstructed, except for 4 references in the `cholesky`, `lu`, `ludcmp`, and `nussinov` benchmarks, for which several loops and statements are synthesized. Out of a total of 107 references, they account for 21.3% of the total data volume. For piecewise-affine traces the problem is exacerbated: multistatement representations are generated for 197 out of 258 references, accounting for 90.7% of the data volume. The average number of statements generated by [15] in this case is 16.4, with a maximum of more than 2000 for `seidel-2d`, a particularly complex input. However, the approach in [15] manages to solve the 4 references in `heat-3d` for which the TRE fails, and decreases maximum loop depth when it generates multiple statements for a single reference.

One application example where minimal reconstruction is desirable is generating equivalent affine versions of non-affine codes, which may then be optimized using an off-the-shelf polyhedral compiler. An example is the optimization of the sparse matrix-vector multiplication by Rodríguez and Pouchet [21]. The approach employed in this work was to i) trace the execution of the irregular SpMV code for a given input matrix; ii) analyze the generated trace using the TRE; iii) generate an affine code that runs the original computation from the TRE output; and iv) generate code using off-the-shelf polyhedral compilers. Generating minimal code in this case proved to be critical to avoid large control overheads in the automatically generated affine codes.

Trace-based code reconstruction has been successfully employed for automatic parallelization. Holewinski et al. [12] use dynamic data dependence graphs derived from sequential execution traces to identify vectorization opportunities. Apollo [14, 23] is a dynamic optimizer which uses linear interpolation and regression to model observed memory accesses. Nearly affine accesses are approximated using two hyperplanes enclosing potentially accessed memory regions, and their convex hull incorporated into the dependence model. Skeleton optimizations are statically built, reducing runtime overhead; and are dynamically selected, instanced, and verified using speculative mechanisms.

To reduce remote memory accesses in NUMA architectures, good data placement is essential. Piccoli et al. [19]

propose a combination of static and dynamic techniques for migrating memory pages with high reuse. A compiler infers affine expressions for array sizes and the reuse of each memory access, and inserts checks to assess the profitability of potential page migrations at runtime. Our proposal can also provide the essential information for data placement in NUMA architectures, either statically after trace-based reconstruction and reconstructed code analysis, or dynamically as a software-based prediction mechanism.

Prior research investigated the problem of designing ad-hoc memory hierarchies for embedded applications. Catthoor et al. [5] proposed a compiler-based methodology to derive optimal memory regions and associated data allocation. Angiolini et al. [1] use a trace-based method that analyzes the access histogram to determine which memory regions to allocate to scratchpad memory [2]. Issenin and Dutt [13] instrument source code to generate annotated memory traces including loop entry and exit points, and use this information to generate affine representations of amenable loops and optimize scratchpad allocation. The TRE can be employed to apply affine techniques for custom memory hierarchy design for applications for which affine analysis of the source code is not feasible. This is of particular interest for IP cores included in embedded devices. It can also be employed to drive memory allocation managers.

6 Concluding Remarks

This work has presented a novel algebraic approach for the construction of formal models of loop codes through the analysis of their memory traces. A Trace Reconstruction Engine (TRE) iteratively builds candidate loops that model increasingly larger portions of the trace by processing the ordered access strides in the memory trace. The mathematical formulation of the problem has been studied, developing methods for the efficient traversal of the solution space. The efficacy of the approach has been demonstrated using both single-domain and piecewise-affine inputs, using the original and tiled PolyBench/C benchmarks, respectively. The experimental results have shown excellent average reconstruction performance, allowing to model traces containing billions of entries in a matter of minutes. These reconstructions are more compact than those generated by alternative approaches, which is critical for code generation. The proposed modeling is widely applicable to a number of different problems, such as automatic code generation and optimization, trace compression, dynamic dependence analysis, memory management, or memory hierarchy design.

Acknowledgments

This research was supported by the Ministry of Economy and Competitiveness of Spain, Project TIN2016-75845-P (AEI/FEDER, EU); NSF grants 1626251, 1409095, 1629129, 1439057, 1213052, 1439021; and a grant from Intel Corp. We gratefully thank Prof. Alain Ketterlin and Prof. Philippe Clauss for providing access to their trace analysis tool.

References

[1] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proc. Int. Conf. Compilers, Archit., Synth. Embed. Syst.*, pages 318–326, 2003.

[2] R. Banakar et al. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proc. 10th Int. Symp. Hardw./Softw. Codesign*, pages 73–78, 2002.

[3] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. ACM Conf. Prog. Lang. Design Impl.*, pages 101–113, 2008.

[5] F. Catthoor et al. *Custom Memory Management Methodology*. Kluwer Academic Publishers, Boston, 1998.

[6] P. Clauss and B. Kenmei. Polyhedral modeling and analysis of memory access profiles. In *Proc. Int. Conf. App.-Spec. Syst. Archit. Procs.*, pages 191–198, 2006.

[7] P. Clauss, B. Kenmei, and J. C. Beyler. The periodic-linear model of program behavior capture. In *Proc. 11th Int. Euro-Par Conf.*, pages 325–335, 2005.

[8] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Proc. 10th Int. Euro-Par Conf.*, pages 292–303, 2004.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Prog.*, 21(6):389–420, 1992.

[10] S. Girbal et al. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Prog.*, 34(3):261–317, 2006.

[11] G. Gupta and S. Rajopadhye. The Z-polyhedral model. In *Proc. 12th ACM Symp. Princ. Pract. Parallel Prog.*, pages 237–248, 2007.

[12] J. Holewinski et al. Dynamic trace-based analysis of vectorization potential of applications. In *Proc. ACM Conf. Prog. Lang. Design Impl.*, pages 371–382, 2012.

[13] I. Issenin and N. Dutt. FORAY-GEN: Automatic generation of affine functions for memory optimizations. In *Proc. Design, Autom. Test Europe Conf. Exp.*, pages 808–813, 2005.

[14] A. Jimborean et al. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *Int. J. Parallel Prog.*, 42(4):529–545, 2014.

[15] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proc. 6th Int. Symp. Code Gen. Opt.*, pages 94–103, 2008.

[16] M. Kobayashi. Dynamic characteristics of loops. *IEEE Trans. Comput.*, 33(2):125–132, 1984.

[17] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM Conf. Prog. Lang. Design Impl.*, pages 190–200, 2005.

[18] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proc. 4th Int. Conf. Comp. Front.*, pages 143–152, 2007.

[19] G. Piccoli et al. Compiler support for selective page migration in NUMA architectures. In *Proc. 23rd Int. Conf. Parallel Archit. Compil. Tech.*, pages 369–380, 2014.

[20] L.-N. Pouchet. PolyBench: The Polyhedral Benchmarking suite. http://polybench.sf.net, 2011. Last accessed: June 2018.

[21] G. Rodríguez and L.-N. Pouchet. Polyhedral modeling of immutable sparse matrices. In *Proc. 8th Int. Works. Polyhedral Compil. Tech.*, 2018.

[22] G. Rodríguez, J. M. Andión, M. T. Kandemir, and J. Touriño. Trace-based affine reconstruction of codes. In *Proc. 14th Int. Symp. Code Gen. Opt.*, pages 139–149, 2016.

[23] A. Sukumaran-Rajam and P. Clauss. The polyhedral model of nonlinear loops. *ACM Trans. Archit. Code Optim.*, 12(4): 48, 2016.

[24] M. Wolfe. More iteration space tiling. In *Proc. ACM/IEEE Supercomput. Conf.*, pages 655–664, 1989.