Trace-Based Affine Reconstruction of Codes

Gabriel Rodríguez
José M. Andión
Dep. Electrónica e Sistemas
Universidade da Coruña
A Coruña, 15071 Spain
{grodriguez,jandion}@udc.es

Mahmut T. Kandemir
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802 USA
kandemir@cse.psu.edu

Juan Touriño
Dep. Electrónica e Sistemas
Universidade da Coruña
A Coruña, 15071 Spain
juan@udc.es

Abstract

Complete comprehension of loop codes is desirable for a variety of program optimizations. Compilers perform static code analyses and transformations, such as loop tiling or memory partitioning, by constructing and manipulating formal representations of the source code. Runtime systems observe and characterize application behavior to drive resource management and allocation, including dependence detection and parallelization, or scheduling. However, the source codes of target applications are not always available to the compiler or runtime system in an analyzable form. It becomes necessary to find alternate ways to model application behavior.

This paper presents a novel mathematical framework to rebuild loops from their memory access traces. An exploration engine traverses a tree-like solution space, driven by the access strides in the trace. It is guaranteed that the engine will find the minimal affine nest capable of reproducing the observed sequence of accesses by exploring this space in a brute force fashion, but most real traces will not be tractable in this way. Methods for an efficient solution space traversal based on mathematical properties of the equation systems which model the solution space are proposed. The experimental evaluation shows that these strategies achieve efficient loop reconstruction, processing hundreds of gigabytes of trace data in minutes. The proposed approach is capable of correctly and minimally reconstructing 100% of the static control parts in PolyBench/C applications. As a side effect, the trace reconstruction process can be used to efficiently compress trace files. The proposed tool can also be used for dynamic access characterization, predicting over 99% of future memory accesses.

CGO'16, March 12–18, 2016, Barcelona, Spain
ACM. 978-1-4503-3778-6/16/03...
http://dx.doi.org/10.1145/2854038.2854056

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; D.4.8 [*Operat. Systems*]: Performance—Modeling and prediction

Keywords Trace analysis, polyhedral optimization, program behavior modeling

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, with static control parts that depend only on the loop index variables and loop independent constants through affine bounds and subscripts, and access and operate on large arrays of data. This is the type of codes that is usually modeled and optimized using the polyhedral approach [4, 7, 14, 17].

Many static and dynamic optimization 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. Even when source code is available, it may not be amenable to static analysis and optimization. Programmers may use complex data and control structures, including code obfuscation techniques [18], that mask the underlying application logic and prevent static analysis and optimization.

This paper presents an exploratory approach for *automatically reconstructing* affine references from a trace of their memory accesses. The exploration engine traverses 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 access strides must be constructed as linear combinations of loop index variables, and only adds a new loop level when no other solution is feasible. 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 efficiently reconstruct the code. 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 proposed approach builds a minimal *equivalent form* using an n-level loop. The generated sequence of references is the same as the original one, but the number of loop levels is not guaranteed to be the same as in the original code. The main contributions of this work are:

- A mathematical framework for the extraction of an affine representation of a given memory trace (Sec. 3), without user intervention or access to source codes or application binaries. Strategies for traversing the solution space towards a minimal representation are provided, including a mechanism that can be used for dynamic prediction of future accesses during runtime.
- A detailed experimental evaluation of the proposed ideas on sequential codes (Sec. 4). Our results show that: (i) the framework can be used to reconstruct large, complex traces, in acceptable time; and (ii) the prediction mechanism anticipates over 99% of the accesses of a linear memory reference.

Besides efficient trace compression, 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 hardware and software prefetching, data placement for locality optimizations, dependence analysis for automatic parallelization, and optimal design of embedded memory systems for locality. These applications are discussed in depth in Sec. 5, along with the related work.

2. Problem Formulation

A program memory trace contains all the memory addresses issued by its entire execution, including multiple loop nests 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 [19]. The address stream generated by each instruction is analyzed separately. A mechanism to detect and extract loop sections in the trace [16, 21] may be used if a single instruction may appear in different loop scopes. The algorithm focuses on the individual reconstruction of each reference. These types of loops can be written as:

$$\begin{array}{l} \texttt{DO } i_1 = 0,\ u_1(\overrightarrow{i}) \\ \quad \vdots \\ \quad \texttt{DO } i_n = 0,\ u_n(\overrightarrow{i}) \\ \qquad V[f_1(\overrightarrow{i})]\dots[f_m(\overrightarrow{i})] \end{array}$$

where $\{u_j, 0 < j \leq n\}$ are affine functions; $\{f_d(i_1, \dots, i_n), 0 < d \leq m\}$ is the set of affine functions that converts a given point in the iteration space of the nest to a point in the data space of V; and $\overrightarrow{i}^k = \{i_1^k, \dots, i_n^k\}^T$ is a column vector which encodes the state of each iteration variable for the k^{th} execution of V. The complete access $V[f_1(\overrightarrow{i})]\dots[f_m(\overrightarrow{i})]$ is abbreviated by $V(\overrightarrow{i})$. Iteration bounds are assumed to be inclusive. Note that this type of loops can be represented as a single $\mathcal{Z}$–polyhedron [9] with dimension n and $2n$ faces. Since f_j is affine, the access can be rewritten as:

$$V[f_1(\overrightarrow{i})]\dots[f_m(\overrightarrow{i})] = V[c_0 + i_1c_1 + \dots + i_nc_n] \quad (1)$$

where V is the base address of the array, c_0 is a constant stride, and each $\{c_j, 0 < j \leq n\}$ is the *coefficient* of the loop index i_j, and must account for the dimensionality of the original array[1]. This is the canonical form into which the method proposed in this paper reconstructs the loop. Note that any sequence of N numbers can be generated using an affine loop which has at most $N - 1$ levels, and thus the algorithm may reconstruct small non-affine loops using affine ones of increased depth.

During the execution of the loop nest, 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 and the size of the accessed data. 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 bounds functions, $u_1(\overrightarrow{i})$, …, $u_n(\overrightarrow{i})$, are known. The values in $\overrightarrow{i}^{k+1}$ can be calculated as follows:

1. An index i_j resets to 0 *iff* all of the following hold:
   - All inner indices are resetting.
   - Either i_j has reached its maximum iteration count, or some inner index has a negative value for its maximum iteration count when i_j increases by one.
2. An i_j increases by one *iff* all of the following hold:
   - All inner indices are resetting.
   - i_j has not reached its maximum iteration count, and all inner indices have non-negative values for their maximum iteration count when i_j increases by one.
3. In any other case, i_j will not change.

These conditions are intuitive and a direct consequence of loop semantics and application control flow.

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 does not change $\Rightarrow \delta_j^k = 0$

[1] 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).

2. i_j is increased by one $\Rightarrow \delta_j^k = 1$
3. i_j is reset to 0 $\Rightarrow \delta_j^k = -i_j^k$

In the following, vector notation will be used for δ:

$$(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^{k}) = \begin{bmatrix} i_1^{k+1} - i_1^k \\ \vdots \\ i_n^{k+1} - i_n^k \end{bmatrix} = \begin{bmatrix} \delta_1^k \\ \vdots \\ \delta_n^k \end{bmatrix} = \overrightarrow{\delta}^k$$

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

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

$$\begin{array}{rlllll} \sigma^k = & V + (c_0 + & c_1 i_1^{k+1} & + \ldots + & c_n i_n^{k+1}) & - \\ & V + (c_0 + & c_1 i_1^{k} & + \ldots + & c_n i_n^{k}) & = \\ = & & c_1 \delta_1^k & + \ldots + & c_n \delta_n^k & = \overrightarrow{c}\,\overrightarrow{\delta}^k \end{array}$$

□

3. Reconstruction Algorithm

The proposed algorithm 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 tree-like space represents a point in the iteration space of the loop. Its root is a trivial loop that generates the first two accesses in the trace. The children of a node in the tree are the indices that can immediately follow the parent in the iteration space. Starting from the root, an exploration engine begins incorporating one access to the reconstructed loop in each step, descending one level into the tree, until it finds a solution for the entire trace or determines that no affine loop is capable of generating the observed sequence of accesses. Each step of the process is conceptually depicted in Fig. 1. The algorithm builds the minimal nest capable of generating the observed access trace[2]. This section models the problem, and develops exploration strategies to efficiently traverse the solution space taking into account its mathematical characteristics.

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

- Vector $\overrightarrow{c} \in \mathbb{Z}^n$ of coefficients of loop indices.
- Matrix $\mathbf{I}^N = [\overrightarrow{\imath}^1 | \ldots | \overrightarrow{\imath}^N] \in \mathbb{Z}^{n \times N}$ of iteration indices.

[2] 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 $V[i]$.

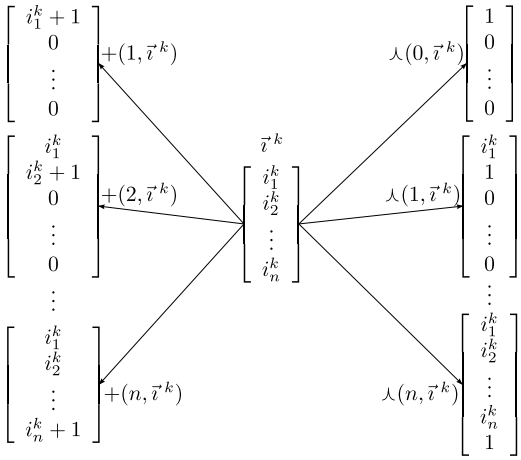


Figure 1. Solution space. For each reconstructed index $\overrightarrow{\imath}^k$, there are $(2n+1)$ possible values for $\overrightarrow{\imath}^{k+1}$. The n alternatives on the left side are obtained using an operation $+(j, \overrightarrow{\imath})$ that increases index i_j by one, and resets to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

- Matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$, and vector $\overrightarrow{w} \in \mathbb{Z}^n$, the bounds matrix and bounds vector, respectively.

The iteration domain $\mathbf{I}^N$ is an integer polyhedron containing the iteration indices $\overrightarrow{\imath}$ such that:

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

where $\mathbf{U}$ is a lower triangular matrix, since no index i_j can depend on the index of an inner loop. Its main diagonal is equal to $\overrightarrow{-1} \in \mathbb{Z}^n$. Its j^{th} row, $U_{(j,:)}$, contains the coefficients of each loop index in the affine bounds function $u_j(\overrightarrow{\imath})$, while w_j contains its independent term. To be a valid solution, $\mathcal{S}_n^N$ has to meet the following requirements:

1. Each consecutive pair of indices $\overrightarrow{\imath}^k$ and $\overrightarrow{\imath}^{k+1}$ must be sequential $\forall k \in [1, N)$ as per Definition 2.1. This condition, which preserves address order, is stronger than simply requiring that the iteration indices stay inside the integer polyhedron delimited by the loop bounds, which can be written by extending Eq. (2) as:

$$\mathbf{U}\mathbf{I}^N + \overrightarrow{w}\mathbf{1}^{1 \times N} \geq \mathbf{0}^{n \times N} \tag{3}$$

2. The observed strides are coherent with the reconstructed ones. Using Lemma 2.2 this can be expressed as:

$$\overrightarrow{c}(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^{k}) = \overrightarrow{c}\,\overrightarrow{\delta}^k = \sigma^k, \forall k \in [1, N)$$

```
1   #define N 32
2   double p[N], A[N][N];
3   for(i = 0; i < N; ++i) {
4      x = A[i][i];
5      for(j = 0; j <= i - 1; ++j)
6         x = x - A[i][j] * A[i][j];
7      p[i] = 1.0 / sqrt(x);
8      for(j = i + 1; j < N; ++j) {
9         x = A[i][j];
10        for(k = 0; k <= i - 1; ++k)
11           x = x - A[j][k] * A[i][k];
12        A[j][i] = x * p[i];
13     }
14  }
```

Figure 2. Source code of the `cholesky` application.

```
1    0x1e2d140
2    0x1e2d140
⋮
30   0x1e2d140
31   0x1e2d240
32   0x1e2d248
33   0x1e2d240
34   0x1e2d248
⋮

88   0x1e2d248
89   0x1e2d340
90   0x1e2d348
91   0x1e2d350
92   0x1e2d340
93   0x1e2d348
94   0x1e2d350
⋮
```

Figure 3. Excerpt of the memory trace generated by the access `A[i][k]` (line 11 of Fig. 2).

The algorithm proceeds iteratively, constructing partial solutions for incrementally larger parts of the trace $\mathcal{A}$. The first partial solution is built as:

$$\mathcal{S}_1^2 = \left\{ \overrightarrow{c} = \left[\sigma^1\right], \mathbf{I}^2 = [0,1], \mathbf{U} = [-1], \overrightarrow{w} = [1] \right\} \quad (4)$$

or, equivalently:

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

Consider the source code of the `cholesky` application from the PolyBench/C 3.2 suite [23] in Fig. 2. For the sake of clarity, in this example we will only focus on the analysis of the access `A[i][k]` in line 11. An excerpt of its memory trace is shown in Fig. 3. The first partial solution, which reconstructs the subtrace $\{a_1 = \texttt{0x1e2d140}, a_2 = a_1\}$, is found to be:

$$\mathcal{S}_1^2 = \left\{ \overrightarrow{c} = [0], \mathbf{I}^2 = [0,1], \mathbf{U} = [-1], \overrightarrow{w} = [1] \right\}$$

Starting from this first partial solution the exploration engine can begin working, gradually increasing its size, until it reaches a solution for the entire trace $\mathcal{A}$. 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:

$$\overrightarrow{c}(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^{k}) = \sigma^k \Rightarrow (\overrightarrow{c}^T\overrightarrow{c})\overrightarrow{\delta}^k = \overrightarrow{c}^T\sigma^k \quad (5)$$

where $(\overrightarrow{c}^T\overrightarrow{c}) \in \mathbb{Z}^{n\times n}$ is the system matrix, and $\overrightarrow{\delta}^k \in \mathbb{Z}^n$ 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, and $\mathbf{I}^{k+1} = \left[\mathbf{I}^k | \overrightarrow{\imath}^{k+1}\right]$. $\mathbf{U}$, $\overrightarrow{w}$, and $\overrightarrow{c}$ remain unchanged. Each of these solutions must be explored independently.
2. The system has no solution generating an index sequential to $\overrightarrow{\imath}^k$, in which case there are three courses of action:
   (a) Increase the dimensionality of the solution (Sec. 3.2).
   (b) Modify the boundary conditions $\mathbf{U}$ and $\overrightarrow{w}$ (Sec. 3.4).
   (c) Discard this branch.

3.1 Solving the Linear Diophantine System

Although the system in Eq. (5) has infinite solutions in the general case, only a few are valid in the context of the affine loop reconstruction, which makes it possible to develop very efficient ad-hoc solution strategies.

Lemma 3.1. *There are at most n valid solutions to the system in Eq. (5). These correspond to indices:*

$$\{+(l, \overrightarrow{\imath}^k) = \left[\ i_1^k\ \ldots\ i_{l-1}^k\ \ \mathbf{i_l^k + 1}\ \ 0\ \ldots\ 0\ \right], 0 < l \leq n\}$$

Proof. If index $\overrightarrow{\imath}^{k+1}$ must be sequential to index $\overrightarrow{\imath}^k$ as per Definition 2.1, then there is a single degree of freedom for $\overrightarrow{\delta}^k$: the position δ_l^k that is equal to 1.

$$\begin{array}{r} \left[\ \ \delta_1^k \ \ \ldots\ \ \delta_{l-1}^k\ \ \delta_l^k\ \ \delta_{l+1}^k\ \ \ldots\ \ \delta_n^k\ \ \right]^T = \\ = \left[\ \ 0\ \ \ldots\ \ 0\ \ 1\ \ -i_{l+1}^k\ \ \ldots\ \ -i_n^k\ \ \right]^T \end{array}$$

Positions $\{i_j, 0 < j < l\}$ will not change between iterations k and $(k+1)$, and therefore $\delta_j^k = 0$; while positions $\{i_j, l < j \leq n\}$ will be reset to 0, and therefore $\delta_j^k = -i_j^k$. □

Taking this result into account, it is possible to find all valid solutions of the system in linear time, $O(n)$, by simply testing the n valid candidates $+(l, \overrightarrow{\imath}^k)$, calculating their associated strides $\hat{\sigma}_l^k = \overrightarrow{c}\,\overrightarrow{\delta}_l^k$, and accepting those solutions with a stride equal to the observed one, $\hat{\sigma}_l^k = \sigma^k$. These are particular solutions of the subtrace $\{a_1, \ldots, a_{k+1}\}$, which will be explored to construct a solution for the entire trace.

Following the `cholesky` example, the next access in the trace to be processed is $a_3 = \texttt{0x1e2d140}$. The engine computes the access stride as $\sigma^2 = a_3 - a_2 = 0$. At this point, a 1-level loop has been constructed and the engine

checks whether $\overrightarrow{i}^3 = +(1, \overrightarrow{i}^2) = [2]$ produces an stride that matches the observed one. The equality $\hat{\sigma}_1^2 = \overrightarrow{c}\,\overrightarrow{\delta}_1^2 = [0]\,[1] = \sigma^2$ holds, and the solution is accepted. The matrix of reconstructed indices is updated, and the algorithm continues processing the trace and updating $\mathbf{I}$ in the same way until it builds $\mathcal{S}_1^{30}$, with $\mathbf{I}^{30} = \left[\begin{array}{cccc} 0 & 1 & \dots & 29 \end{array}\right]$. At this point, the observed stride changes to:

$$\sigma^{30} = a_{31} - a_{30} = \texttt{0x1e2d240} - \texttt{0x1e2d140} = 256$$

The constructed loop with $\overrightarrow{c} = [0]$ cannot produce a stride different from 0. As such, the subtrace $\{a_1, \dots, a_{31}\}$ cannot be generated with an affine access enclosed in a 1-level loop and the dimensionality of the current solution $\mathcal{S}_1^{30}$ must be increased to build $\mathcal{S}_2^{31}$.

3.2 Increasing Solution Dimensionality

Let $\mathcal{S}_n^k = \{\overrightarrow{c}, \mathbf{I}^k, \mathbf{U}, \overrightarrow{w}\}$ be a partial solution for the subtrace $\{a_1, \dots, a_k\}$. If no valid index $\{+(l, \overrightarrow{i}^k), 0 < l \leq n\}$ provides $\hat{\sigma}_l^k = \sigma^k$, it may be because a loop index which had not appeared before is increasing in access $(k+1)$. This can cause σ^k to be unrepresentable either as a linear combination of the loop coefficients $\overrightarrow{c}$, or as an index sequential to $\overrightarrow{i}^k$. It is possible to generate a valid partial solution $\mathcal{S}_{n+1}^{k+1}$ from $\mathcal{S}_n^k$ by enlarging the dimensionality of the current solution components. There are $(n+1)$ such potential solutions, corresponding to the indices $\{\curlywedge(p, \overrightarrow{i}^k), 0 \leq p \leq n\}$. For each insertion position p of the newly discovered loop, the set of indices $\mathbf{I}^{k+1} \in \mathbb{Z}^{(n+1)\times(k+1)}$, is built as:

$$\mathbf{I}^{k+1} = \left[\begin{array}{c|c} \mathbf{I}^k_{(1:p,:)} & \overrightarrow{i}^{k+1} \\ \mathbf{0}^{1\times k} & \\ \mathbf{I}^k_{(p+1:n,:)} & \end{array}\right]$$

where a 0 in position p has been added to each index in $\mathbf{I}^k$, and a new column $\overrightarrow{i}^{k+1} = \curlywedge(p, \overrightarrow{i}^k)$ has been added to the matrix. The coefficient c'_p associated with the new loop index can be derived from Eq. (5):

$$\overrightarrow{c}\,\overrightarrow{\delta}^k = \left[\begin{array}{ccccc} \dots & c_p & \boldsymbol{c'_p} & c_{p+1} & \dots \end{array}\right] \left[\begin{array}{c} \vdots \\ 0 \\ 1 \\ -i_p^k \\ \vdots \end{array}\right] = \sigma^k \Rightarrow$$

$$c'_p = \sigma^k + \sum_{r=p+1}^{n} i_r^k c_r$$

$\mathbf{U}$ and $\overrightarrow{w}$ are updated as described in Sec. 3.4 to reflect any new information available. If no solution is found for the boundary conditions, then this branch is discarded. Note that there must be a practical limit to the maximum acceptable solution size, as in the general case any trace $\{a_1, \dots, a_N\}$ can be generated using at most $N-1$ affine nested loops. To ensure that a minimal solution, in terms of the dimensionality of the generated $\mathcal{Z}$–polyhedron, is found, the solution space should be traversed in a *breadth-first* fashion.

Revisiting the `cholesky` example, there are two possible insertion points for the new loop in $\mathcal{S}_2^{31}$. As the most common situation is that newly discovered loops are outer than the previously known ones, it explores $p = 0$ first. The new loop coefficient vector and index matrix are calculated as:

$$c'_0 = \sigma^{30} + i_1^{30} c_1 = 256 + 0 \cdot 29 \Rightarrow \overrightarrow{c} = \left[\begin{array}{cc} \mathbf{256} & 0 \end{array}\right]$$

$$\mathbf{I}^{31} = \left[\begin{array}{ccc|c} 0 & \dots & 0 & \mathbf{1} \\ 0 & \dots & 29 & \mathbf{0} \end{array}\right]$$

The traversal of the solution space continues. The next observed stride is $\sigma^{31} = a_{32} - a_{31} = 8$. No increase of the currently found loop indices produces such stride:

$$\begin{cases} \hat{\sigma}_1^{31} = \overrightarrow{c}\,\overrightarrow{\delta}_1^{31} = \left[\begin{array}{cc} 256 & 0 \end{array}\right]\left[\begin{array}{cc} 1 & 0 \end{array}\right]^T = 256 \\ \hat{\sigma}_2^{31} = \overrightarrow{c}\,\overrightarrow{\delta}_2^{31} = \left[\begin{array}{cc} 256 & 0 \end{array}\right]\left[\begin{array}{cc} 0 & 1 \end{array}\right]^T = 0 \end{cases}$$

Hence, the solution must grow to $\mathcal{S}_3^{32}$. Now there are three different insertion points. The first two yield the following coefficient vectors:

$$\begin{cases} p = 0 \Rightarrow \overrightarrow{c} = \left[\begin{array}{ccc} \mathbf{264} & 256 & 0 \end{array}\right] \\ p = 1 \Rightarrow \overrightarrow{c} = \left[\begin{array}{ccc} 256 & \mathbf{8} & 0 \end{array}\right] \end{cases}$$

As soon as the first points are explored in these branches, the engine will find that this partial solution does not match the remainder of the trace either. For the sake of simplicity, let us assume that the engine has been configured to explore up to 3-level loops before discarding a branch, and thus it will not try to build $\mathcal{S}_4^{33}$. Rather, it will continue the exploration on the third possible insertion point:

$$p = 2 \Rightarrow \overrightarrow{c} = \left[\begin{array}{ccc} 256 & 0 & \mathbf{8} \end{array}\right]$$

At this point the engine has correctly recognized the coefficients of the three levels of the original nest. It generates the new index matrix:

$$\mathbf{I}^{32} = \left[\begin{array}{ccc|c} & \mathbf{I}_{(1:2,:)} & & \overrightarrow{i}^{32} \\ \mathbf{0} & \dots & \mathbf{0} & \end{array}\right] = \left[\begin{array}{cccc|c} 0 & \dots & 0 & 1 & \mathbf{1} \\ 0 & \dots & 29 & 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{\dots} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{array}\right]$$

For the sake of simplicity, this section does not discuss the calculations associated to loop bounds. These will be detailed in Sec. 3.4.

3.3 Branch Priority

The approach proposed above is capable of efficiently finding the relevant solutions of the linear diophantine system for each address of the trace, but can still produce a large number of potential solutions that will be discarded when processing the remaining addresses in the trace. In the general case, the time for exploring the entire solution space of a trace containing N addresses generated by n loops would

be $O(n^N)$. Consequently, exploring all branches with no particular order could take a very long time. To guide the traversal of the solution space, consider the column vector $\overrightarrow{\gamma}^k \in \mathbb{Z}^n$ defined as:

$$\overrightarrow{\gamma}^k = \mathbf{U}\overrightarrow{\imath}^k + \overrightarrow{w} \tag{6}$$

Lemma 3.2. *Each element $\gamma_j^k \in \overrightarrow{\gamma}^k$ indicates how many more iterations of index i_j are left before it resets under bounds $\mathbf{U}$, $\overrightarrow{w}$.*

Proof. γ_j^k is equal to the value of the upper bound of the loop in i_j minus the current value of i_j:

$$\gamma_j^k = \underbrace{\mathbf{U}_{(j,:)}\overrightarrow{\imath}^k + w_j}_{w_j + u_{j,1}i_1 + \ldots + u_{j,(j-1)}i_{(j-1)} - i_j} = u_j(\overrightarrow{\imath}) - i_j$$

By construction of the canonical loop form, the step of all loops is 1. Therefore, γ_j^k is equal to the number of iterations of loop i_j before $i_j > u_j(\overrightarrow{\imath})$. □

This result suggests that, assuming that $\mathbf{U}$ and $\overrightarrow{w}$ are accurate, the most plausible value for the next index is $\overrightarrow{\imath}^{k+1} = +(l, \overrightarrow{\imath}^k)$, where l is the position of the innermost positive element of $\overrightarrow{\gamma}^k$. The correctness of this prediction can be assessed by comparing the predicted stride $\hat{\sigma}_l^k$ with the observed σ^k. Note that using $\overrightarrow{\gamma}^k$ as described above guarantees consistency with the boundary conditions in Eq. (2), which further improves the efficiency of the approach by saving calculations.

3.4 Calculating Loop Bounds

So far the calculation of the boundary conditions, $\mathbf{U}$ and $\overrightarrow{w}$, has been overlooked. As before, assume that the algorithm has already identified a partial solution $\mathcal{S}_n^k = \{\overrightarrow{c}, \mathbf{I}^k, \mathbf{U}, \overrightarrow{w}\}$. Upon processing access a_{k+1} the algorithm will try to explore the branch which increments the index i_l corresponding to the innermost positive element of $\overrightarrow{\gamma}^k$, as described in Sec. 3.3. However, it might happen that the calculated stride for the selected branch does not match the observed stride, i.e., $\hat{\sigma}_l^k \neq \sigma^k$. A different candidate index $i_{l'}$ will have to be generated as described in Sec. 3.1, but the resulting $+(l', \overrightarrow{\imath}^k)$ will not be sequential to $\overrightarrow{\imath}^k$ under the current bounds $\mathbf{U}$ and $\overrightarrow{w}$. In this scenario it is necessary to generate new bounds $\mathbf{U}'$ and $\overrightarrow{w}'$. These can be found by solving the system in Eq. (3):

$$\mathbf{U}'\mathbf{I}^{k+1} + \overrightarrow{w}'\mathbf{1}^{1\times(k+1)} \geq \mathbf{0}^{n\times(k+1)} \tag{7}$$

If the system is inconsistent, then the generated iteration space is not a polytope, and the solution is not valid. If the system has solutions, then it will be overdetermined in the general case. Matrix $\mathbf{U}'$ and vector $\overrightarrow{w}'$ are only partially unknown: the only rows that may vary with respect to $\mathbf{U}$ and $\overrightarrow{w}$ are those corresponding to loop indices $\{i_j, l \leq j \leq n\}$, since the outer variables cannot be affected by the inner, unscoped ones. As such, their first $(l-1)$ rows are known.

First, $\overrightarrow{w}'$ is calculated. The first $(l-1)$ positions are already known and are the same as those in $\overrightarrow{w}$. To calculate the remaining positions $\{w'_j, l \leq j \leq n\}$, consider the set of $(n-l+1)$ reduced systems:

$$\mathbf{U'}_{(j,:)}\overrightarrow{\imath} + w'_j = u'_j(\overrightarrow{\imath}) = 0 \tag{8}$$

where $\mathbf{U'}_{(j,:)}$ and w'_j are unknowns, and $\overrightarrow{\imath} \in \mathbf{I}^{k+1}$ may be any vector such that $u'_j(\overrightarrow{\imath}) = 0$.

Lemma 3.3. *In order to solve Eq (8), it is always possible to choose an index $\overrightarrow{z} \in \mathbf{I}^{k+1}$ of the form:*

$$\overrightarrow{z} = [0, \ldots, 0, z_j, \ldots, z_n]^T$$

Replacing it in the previous equation and taking into account that $\mathbf{U}'$ is a lower triangular matrix with main diagonal equal to $\overrightarrow{-1} \in \mathbb{Z}^n$, Eq. (8) is reduced to $w'_j = z_j$. The only candidate vector which fulfills $u'_j(\overrightarrow{z}) = 0$ is $\overrightarrow{Z}$ such that its Z_j value is maximum, i.e., $(\nexists \overrightarrow{z}, z_j > Z_j)$.

Proof. At least one candidate vector, $\overrightarrow{0} \in \mathbb{Z}^n$, is guaranteed to exist, by construction of the canonical loop form. If a $\overrightarrow{z}$ is chosen such that $w'_j = z_j < Z_j$, Eq. (3) would not hold, at least, for index $\overrightarrow{Z}$:

$$u'_j(\overrightarrow{Z}) = \mathbf{U'}_{(j,:)}\overrightarrow{Z} + w'_j = -Z_j + z_j < 0$$

□

Intuitively, we are operating on the integer polyhedron $\mathbf{I}^k$, selecting the vertex point such that its value along dimension j is maximum. Applying Lemma 3.3 to the `cholesky` example, the bounds vector is calculated as:

$$\overrightarrow{w}' = \begin{bmatrix} 1 & 29 & 0 \end{bmatrix}^T$$

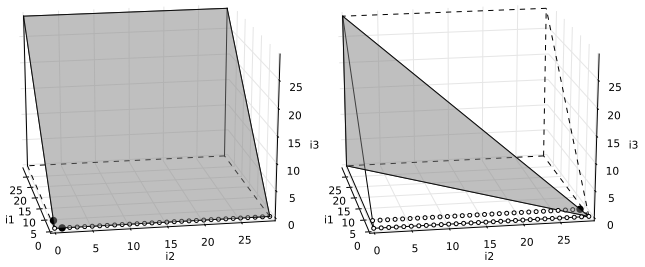
Once $\overrightarrow{w}'$ is calculated, the unknown rows $\mathbf{U'}_{(l:n,:)}$ can be calculated by reducing the original system in Eq. (7) to $(n-l+1)$ equation systems of the form:

$$\mathbf{U'}_{(j,:)}\mathbf{i} + w'_j\mathbf{1}^{1\times(j-1)} = \mathbf{0}^{1\times(j-1)} \tag{9}$$

where $\mathbf{i} \in \mathbb{Z}^{n\times(j-1)}$ is a matrix of columns extracted from $\mathbf{I}^{k+1}$ such that $\big(\forall \overrightarrow{\imath} \in \mathbf{i}, u'_j(\overrightarrow{\imath}) = 0\big)$. $\mathbf{i}$ must contain $(j-1)$ columns as there are $(j-1)$ unknowns in $\mathbf{U'}_{(j,:)}$. As in the calculation of w'_j, each column in $\mathbf{i}$ has to be chosen such that its i_j value is maximum for a specific combination of its indices $(i_1, \ldots, i_{j-1})$. Intuitively, we are building the constraint $u_j(\overrightarrow{\imath}) \geq 0$ from selected points which belong to its associated face in polyhedron $\mathbf{I}^k$. Applying Eq. (9) to index i_3 in the `cholesky` example:

$$\mathbf{U'}_{(3,:)}\mathbf{i} + w'_3\mathbf{1}^{1\times2} = \mathbf{0}^{1\times2} \Rightarrow$$

$$\begin{bmatrix} u_{3,1} & u_{3,2} & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix} \Rightarrow$$



(a) Discovery of index i_3 and associated bounds when processing a_{31}.

(b) Recalculation of i_2 bounds when processing access a_{88}.

Figure 4. Evolution of predicted polyhedron faces through the reconstruction process of access `A[i][k]` in `cholesky`. Faces associated to the row of $\mathbf{U}$ being recalculated are shaded. Edges of the previously predicted polyhedron are dashed. Iteration points already discovered are hollow. Points used to buid matrix $\mathbf{i}$ during the calculation of the face are marked in black.

$$\left[\begin{array}{cc} u_{3,2} & (u_{3,1}-1) \end{array}\right] = \left[\begin{array}{cc} 0 & 0 \end{array}\right] \Rightarrow \left\{\begin{array}{l} u_{3,1} = 1 \\ u_{3,2} = 0 \end{array}\right.$$

So the calculated $\mathbf{U}'$ is:

$$\mathbf{U}' = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ \mathbf{1} & \mathbf{0} & -1 \end{bmatrix}$$

The calculated bounds are shown in Fig. 4(a). Continuing the example, $\overrightarrow{\gamma}^{32} = [0, 29, 0]^T$, and the engine predicts $\overrightarrow{i}^{33} = [1, \mathbf{1}, 0]^T$ which generates a stride that matches the observed one. $\overrightarrow{\gamma}^{33} = [0, 28, 1]^T$ and the engine predicts $\overrightarrow{i}^{34} = [1, 1, \mathbf{1}]^T$ which also generates a stride that matches the observed one. This process continues, alternately incorporating iterations of i_2 and iterations of i_3, until the engine incorporates access a_{88} to the solution, with index $\overrightarrow{i}^{88} = [1, 28, \mathbf{1}]^T$. At this point, $\overrightarrow{\gamma}^{88} = [0, 1, 0]^T$, and the engine predicts an iteration of i_2, with $\hat{\sigma}_2^{88} = -8$. However, $\sigma^{88} = a_{89} - a_{88} = 248$. The engine defaults to the brute force mode, calculating the strides for each of the currently known indices (see Sec. 3.1):

$$\left\{\begin{array}{l} \hat{\sigma}_1^{88} = \overrightarrow{c}\,\overrightarrow{\delta}_1^{88} = \left[\begin{array}{ccc} 256 & 0 & 8 \end{array}\right]\left[\begin{array}{ccc} 1 & -28 & -1 \end{array}\right]^T = \mathbf{248} \\ \hat{\sigma}_3^{88} = \overrightarrow{c}\,\overrightarrow{\delta}_3^{88} = \left[\begin{array}{ccc} 256 & 0 & 8 \end{array}\right]\left[\begin{array}{ccc} 0 & 0 & 1 \end{array}\right]^T = 8 \end{array}\right.$$

The engine explores the branch with $\overrightarrow{i}^{89} = [2, 0, 0]$, and the loop bounds have to be updated:

$$\overrightarrow{w}' = \left[\begin{array}{ccc} \mathbf{2} & 29 & 0 \end{array}\right]$$

The first and third rows of $\mathbf{U}'$ do not change. For the second, the following system is solved:

$$\mathbf{U}'_{(2,:)}\mathbf{i} + w'_2\mathbf{1}^{1\times 1} = \mathbf{0}^{1\times 1} \Rightarrow$$

$$\left[\begin{array}{ccc} u_{2,1} & -1 & 0 \end{array}\right]\begin{bmatrix} 1 \\ 28 \\ 1 \end{bmatrix} + [29] = [0] \Rightarrow u_{2,1} = -1$$

and the new bounds matrix:

$$\mathbf{U}' = \begin{bmatrix} -1 & 0 & 0 \\ \mathbf{-1} & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

This calculation is illustrated in Fig. 4(b). The engine has now collected all the information that it needs to solve the problem. From this point on, the engine will keep incorporating elements in the trace to the solution, with $\overrightarrow{\gamma}$ accurately predicting all remaining iterations, until it reaches the end of the trace having reconstructed the following terms:

$$\left.\begin{array}{l} \overrightarrow{c} = \left[\begin{array}{ccc} 256 & 0 & 8 \end{array}\right] \\ \mathbf{U} = \begin{bmatrix} -1 & 0 & 0 \\ -1 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix} \\ \overrightarrow{w} = \left[\begin{array}{ccc} 29 & 29 & 0 \end{array}\right] \end{array}\right\} \begin{array}{l} \texttt{DO } i_1 = 0,\ 29 \\ \quad \texttt{DO } i_2 = 0,\ 29 - i_1 \\ \quad\quad \texttt{DO } i_3 = 0,\ i_1 \\ \quad\quad\quad a_1 + 256i_1 + 8i_3 \end{array}$$

Note that this reconstruction method does not regenerate the constant term c_0 in Eq. (1), and assumes the base address of the access to be $V' = a_1$. This is not a problem for any practical application of the extracted loop information, as the set of accessed points is identical to that of the original, potentially non-canonical loop.

3.5 Algorithm

Algorithm 1 presents the pseudocode of the `Extract()` function which implements the proposed approach. The recursive solution is not practical for a real implementation, but clearly illustrates the idea. The computations to calculate the new loop insertions described in Sec. 3.2 are encapsulated in a `Grow()` function, shown as Algorithm 2. The reconstruction starts by calling `Extract()` with the initial $\mathcal{S}_1^2$ defined in Eq. (4). In the worst case, when no access is correctly predicted using $\overrightarrow{\gamma}$, the algorithm uses the brute force approach ($O(n^N)$). In the best case every access is correctly predicted ($O(N)$).

4. Experimental Results

The proposed algorithm has been implemented in Python and used to extract codes for different affine kernels. This section analyzes the behavior of the reconstruction algorithm in order to assess the feasibility of the proposed approach. The reconstruction algorithm was run with traces generated by the PolyBench/C 3.2 suite [23]. It includes 30 applications from domains such as linear algebra, stencil codes, and data mining. The target was the traces generated by the static control parts of these applications (enclosed within `scop` pragmas). These were split into the subtraces generated by their different instructions and stored in memory before being processed. The "standard" problem size was used, generating traces ranging from 6 million references for `jacobi-1D`

Algorithm 1: Pseudocode of `Extract()`

Input: the execution trace, $\mathcal{A}$, and a partial solution $\mathcal{S} = \{\overrightarrow{c}, \mathbf{I}, \mathbf{U}, \overrightarrow{w}\}$
Output: a global solution or $None$ if no solution found

$k = \#$columns of $\mathbf{I}$;
while $k < len(\mathcal{A}) - 1$ **do**
  $\sigma = a_{k+1} - a_k$;
  `// Try to use` $\overrightarrow{\gamma}$ `(Sec. 3.3)`
  calculate $\overrightarrow{\gamma} = U\overrightarrow{\imath}^k + \overrightarrow{w}$;
  calculate predicted stride $\hat{\sigma}_l = \overrightarrow{c}\,\overrightarrow{\delta}_l$;
  if $\hat{\sigma}_l = \sigma$ **then**
    $\mathbf{I} = [\mathbf{I}| + (l, \overrightarrow{\imath}^k)]$;
    k = k+1;
    continue;
  end
  `// Brute force approach (Sec. 3.1)`
  for *l*=*n down to* 1 **do**
    calculate $\hat{\sigma}_l = \overrightarrow{c}\,\overrightarrow{\delta}_l$;
    if $\hat{\sigma}_l = \sigma$ **then**
      $\mathbf{I}' = [\mathbf{I}| + (l, \overrightarrow{\imath}^k)]$;
      $\{\mathbf{U}', \overrightarrow{w}'\}$ = update bounds; `// Sec. 3.4`
      if $\{\overrightarrow{c}, \mathbf{I}', \mathbf{U}', \overrightarrow{w}'\}$ *is linear* **then**
        $\mathcal{S}' = Extract(\{\overrightarrow{c}, \mathbf{I}', \mathbf{U}', \overrightarrow{w}'\}, \mathcal{A})$;
        if $\mathcal{S}' \neq None$ **then** return $\mathcal{S}'$;
      end
    end
  end
  `// Add loop (Sec. 3.2)`
  for p=0 *to* n **do**
    $\mathcal{S}' = Extract(Grow(\mathcal{S}, p), \mathcal{A})$;
    if $\mathcal{S}' \neq None$ **then** return $\mathcal{S}'$;
  end
  return $None$;
end
return $\mathcal{S}$;

(150 MB in disk) to 12.9 billion references for `3mm` (270 GB). The number of references in each kernel varies between 3 for `trmm` and 92 for `fdtd-apml`. Each execution was performed on an Intel Xeon E5-2660 Sandy Bridge 2.20 Ghz node, with 64 GB of RAM.

Fig. 5 shows trace sizes and processing times. 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 `jacobi-1D`, a stencil computation which only accesses small 1-dimensional arrays. Two loops generate all traces, but the outer one iterates only once per each 10.000 iterations of the inner one. As a result, the reconstruction process can be largely streamlined: the trace contains blocks of 10.000 elements separated by the same stride, which can be recognized in a single step using $\overrightarrow{\gamma}$ as a predictor. Its 6 million accesses are sequentially processed in 0.2 seconds. On the opposite end, `dynprog`, which emits

Algorithm 2: Pseudocode of `Grow()` (Sec. 3.2)

Input: the partial solution $\mathcal{S} = \{\overrightarrow{c}, \mathbf{I}, \mathbf{U}, \overrightarrow{w}\}$, and the insertion point x
Output: modified partial solution with a new loop in position x, or $None$ if the insertion point generates a nonlinear solution

`// Insert a new row and column in` $\mathbf{U}$
$\mathbf{U} = \left[\begin{array}{c|c|c} \mathbf{U}_{(1:x,1:x)} & \mathbf{0}^{x\times 1} & \mathbf{U}_{(1:x,x+1:n)} \\ 0\ldots 0 & -1 & 0\ldots 0 \\ \mathbf{U}_{(x+1:n,1:x)} & \mathbf{0}^{(n-x)\times 1} & \mathbf{U}_{(x+1:n,x+1:n)} \end{array}\right]$;
`// Insert a new element in` $\overrightarrow{w}$
$\overrightarrow{w} = \left[\overrightarrow{w}_{(1:x)}|0|\overrightarrow{w}_{(x+1:n)}\right]$;
`// Insert new index into` $\mathbf{I}$
$\mathbf{I} = \left[\begin{array}{c|c} \mathbf{I}_{(1:x,:)} & \\ 0\ldots 0 & \lambda(x, \overrightarrow{\imath}^k) \\ \mathbf{I}_{(x+1:n,:)} & \end{array}\right]$;
$\{\mathbf{U}', \overrightarrow{w}'\}$ = update bounds; `// Sec. 3.4`
$\overrightarrow{c} = [\overrightarrow{c}_{(1:x)}|c_x|\overrightarrow{c}_{(x+1:n)}]$;
if $\{\overrightarrow{c}, \mathbf{I}, \mathbf{U}, \overrightarrow{w}\}$ *is not linear* **then** return $None$;
return $\{\overrightarrow{c}, \mathbf{I}, \mathbf{U}, \overrightarrow{w}\}$;

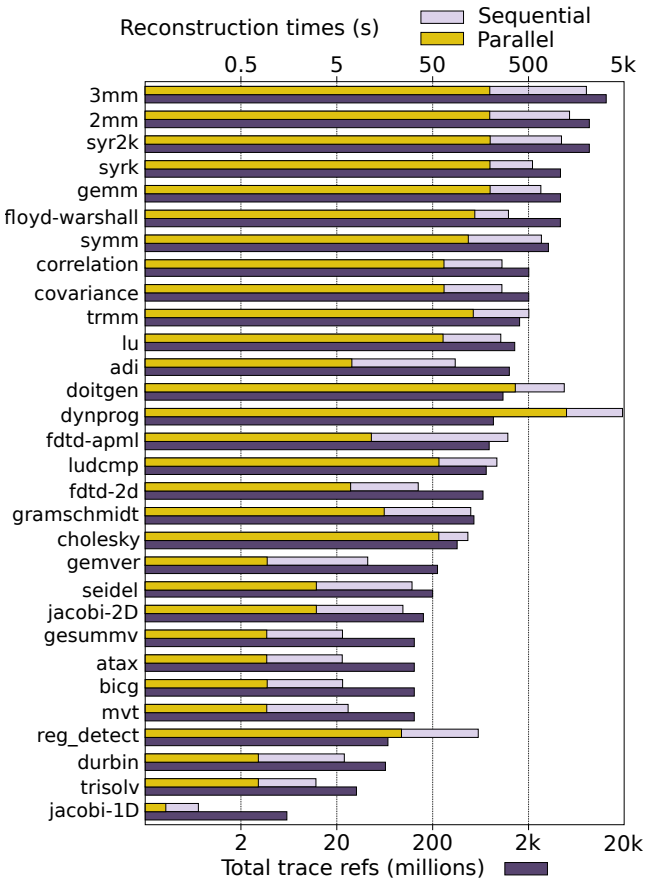


Figure 5. Reconstruction times (upper axis) and trace sizes (lower axis) for the PolyBench/C benchmarks, ordered by trace size. Axes are logarithmic. Since the subtraces of a kernel are independent they can be reconstructed in parallel, achieving an average speedup of 5.6x.

Trace	%	Trace	%	Trace	%
3mm	0.02	lu	0.11	seidel	0.00
2mm	0.04	adi	0.01	jac-2D	0.00
syr2k	0.02	doit.	0.58	gesum.	25.01
syrk	0.05	dynp.	0.00	atax	25.00
gemm	0.05	fdtd-a.	24.21	bicg	25.00
floyd	0.00	lud.	0.66	mvt	12.50
symm	0.13	fdtd-2d	0.01	reg_d.	2.07
corr.	0.67	grams.	0.58	durbin	100
covar.	0.37	chol.	0.58	trisolv	100
trmm	0.00	gemv.	21.43	jac-1D	100

Table 1. Percentage of trace reconstructed after 48h without $\overrightarrow{\gamma}$ prediction.

Trace	%	Trace	%	Trace	%
3mm	99.93	lu	99.99	seidel	99.97
2mm	99.91	adi	97.42	jac-2D	99.97
syr2k	99.88	doit.	99.88	gesum.	74.93
syrk	99.85	dynp.	99.85	atax	74.94
gemm	99.90	fdtd-a.	75.70	bicg	74.94
floyd	99.90	lud.	99.99	mvt	87.43
symm	99.90	fdtd-2d	97.14	reg_d.	99.79
corr.	99.72	grams.	99.72	durbin	99.88
covar.	99.90	chol.	99.99	trisolv	99.89
trmm	99.99	gemv.	78.51	jac-1D	98.00

Table 2. Percentage of trace accesses predicted by $\overrightarrow{\gamma}$.

858 million references, is the one processed at the slowest rate. It features a 4-level loop nest where the largest block of single-strided accesses contains only 48 references. As such, the number of decision steps taken by the algorithm is much larger. While in the slowest case the engine is capable of processing 170.000 references per second, in the fastest one this figure goes up to 33 million (194x faster).

A second set of experiments was run deactivating $\overrightarrow{\gamma}$ prediction. In this case, the engine must explore all potentially correct branches as indicated in Sec. 3.1. All subtraces were processed in parallel. The recognition was run for 48 hours, at which point the unreconstructed subtraces were considered intractable for practical purposes. Table 1 summarizes the results. For most codes only the smallest subtraces were recognized, accounting for less than 1% of the total trace. `fdtd-apml`, `gemver`, `gesummv`, `atax`, `bicg`, and `mvt` contain large single-strided subtraces, which are recognized as a single block. `durbin` and `trisolv` have subtraces of 8 million references, each of which is reconstructed in 47 hours. `jacobi-1D` has subtraces of only 1 million references.

The usability of the engine as an online predictor was also evaluated. Table 2 shows the percentage of predicted accesses. The engine is made to predict the next access in the trace. When the prediction matches the address, it is counted as a hit. If the prediction is a mismatch, it is counted as a miss, the solution is modified to adapt to the new observation and a new prediction is made for the next access. For most applications, $\overrightarrow{\gamma}$ predicted above 95% of the issued references. Exceptions are, again, `fdtd-apml`, `gemver`, `gesummv`, `atax`, `bicg`, and `mvt`. Note how their numbers are almost complementary to those in Table 1. The reason is that most unpredicted accesses were issued by single-strided references. These are not handled by $\overrightarrow{\gamma}$ since it cannot operate before $\overrightarrow{w}$ is calculated, and this will never happen for 1-level loops, which generate the types of traces that are tractable by the algorithm without $\overrightarrow{\gamma}$ guidance. However, since these loops are single-strided, the engine is capable of correctly predicting which addresses will be issued in the future by simply assuming that the only known loop will iterate. As such, the combined prediction rate is above 99% for all the codes we tested.

Regarding memory requirements, the exploration engine needs to store, at least, $\overrightarrow{c}$, $\overrightarrow{w}$, $\mathbf{U}$, and selected indices of $\mathbf{I}$[3]. In addition to these, some memory is consumed by backtracking points used to efficiently implement the recursion in Algorithm 1. The total memory requirements for subtraces in our experimental set-up vary between 48 bytes and 44 KB.

5. Related Work and Applications

Not many works, to the best of the authors' knowledge, have explored the reconstruction of loop codes from their memory access traces. Most of them have done it as a means to pursue a particular optimization. This section organizes related work according to their ultimate goal, also discussing the potential applications of the exploration engine proposed in this paper.

Clauss et al. [6] characterized program behavior using polynomial piecewise periodic and linear interpolations separated into adjacent program phases to reduce function complexity. The model can be recursively applied, interpreting coefficients of the periodic interpolation as traces in themselves. 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. It uses a stack of terms. When a new term is pushed, it searches for triplets of terms that can be rewritten as a loop. This approach is capable of affinely representing entire traces, generating imperfectly nested loops without the need for pre- or post-processing steps. A bound must be imposed on maximal instruction interleaving in order for the approach to be practical. In contrast, in our approach it is necessary to analyze the trace first, extracting the individual memory references and their loop scopes. These isolated memory address streams are the input to the algorithm presented in Sec. 3. In terms of optimality, our approach is guaranteed to find the minimal solution for each stream,

[3] The only indices that need to be stored during the reconstruction process are those $\{\overrightarrow{i} \in \mathbf{I}, \exists j, 0 < j \leq n, u_j(\overrightarrow{i}) = 0\}$, i.e., points on a face of the iteration polyhedron. These are used for calculating $\overrightarrow{w}$ and $\mathbf{U}$ (see Sec. 3.4).

should it exist, while the approach by Ketterlin and Clauss, which merges triplets in a greedy way and never backtracks, may find non-minimal solutions. For instance, it compresses the `cholesky` example used throughout the paper using 11 different 1- to 4-level loops, containing 12 imperfectly nested references. For $N = 128$, our proposed approach still reconstructs a single 3-level loop with the same structure as for the smaller trace, while the approach by Ketterlin and Clauss employs 11 loops of up to 6 levels.

One potential use of the exploration engine is cache prefetching. In order to improve on the one block lookahead scheme [24], Baer and Chen [2] use a prediction table and lookahead program counter to preload regular accesses which correctly predict the stride of the innermost loop. Iacobovici et al. [11] propose a prefetcher capable of supporting up to four distinct strides. In contrast, our approach is capable of supporting an unlimited amount of strides, as well as variable trip counts. However, hardware prefetching using our loop reconstruction mechanism requires memory space to store at least the values of $\mathbf{U}$ and $\overrightarrow{w}$ for each loop, as well as $\overrightarrow{c}$ for each access instruction. The required space depends on the maximum nesting level supported. Other prefetchers integrated in the memory controller [25] could also benefit from a hardware implementation of our recognition engine to improve prediction accuracy.

To reduce remote memory accesses in NUMA architectures, good data placement is essential. kMAF [8] improves data locality by dynamically analyzing page faults of running applications and migrating threads and memory pages consequently. It is engineered into the virtual memory implementation of the operating system. Piccoli et al. [22] propose a combination of static and dynamic techniques for migrating memory pages predicted to be frequently reused. A compiler infers affine expressions for array sizes and the reuse of each memory access enclosed in loops, 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.

Trace-based code reconstruction is also useful for automatic parallelization. Holewinski et al. [10] use dynamic data dependence graphs derived from sequential execution traces to identify vectorization opportunities. Jimborean et al. [13] propose a dynamic mechanism for detecting data dependences using interpolated linear functions to approximate observed memory accesses to guide speculative parallelization. Mendis et al. [20] employ the binary and data collected from several executions of an application to rewrite computational kernels into a DSL by building a forest of expressions and translating it into an optimized version. Similar systems can be constructed using the proposed exploration engine, capable of analyzing dependences without the need for compiler support.

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 [3]. Issenin and Dutt [12] 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 SPM allocation. Our trace-based reconstruction approach 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, commonly included in embedded devices. It can also be employed to drive scratchpad allocation managers.

6. Concluding Remarks

This work has explored the affine reconstruction of loop codes from their memory traces, focusing on one instruction at a time. Large traces are processed in a matter of minutes without user intervention or access to source or binary codes. The proposed methodology has applications such as trace compression/storage/communication, dynamic parallelization, or memory placement and memory hierarchy design. The problem has been formulated as the exploration of a tree-like solution space, in which each node represents a point in the iteration space of the loop. The mathematical relationship amongst the nodes has been established, and the system of equations that governs the trace-based reconstruction of the code has been defined. Afterwards, methods for efficient traversal of this solution space have been proposed. The experimental evaluation has shown good performance and accuracy in the reconstruction of affine codes. Furthermore, it has been shown that the problem is not trivially tractable without the proposed optimizations.

Acknowledgments

This research was partially supported by the Ministry of Economy and Competitiveness of Spain (Project TIN2013-42148-P), co-funded by FEDER Funds of the European Union (80%); NSF grants 1213052, 1439021, 1409095, and 1526750; and a grant from Intel. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions which greatly helped improve the quality of this paper.

References

[1] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES*, pages 318–326, San Jose, CA, USA, 2003. doi:10.1145/951710.951751.

[2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, SC*, pages 176–186, Albuquerque, NM, USA, 1991. doi:10.1145/125826.125932.

[3] R. Banakar, S. Steinke, L. Bo-Sik, M. Balakrishnan, and P. Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign, CODES*, pages 73–78, Estes Park, CO, USA, 2002. doi:10.1145/774789.774805.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, PLDI*, pages 101–113, Tucson, AZ, USA, 2008. doi:10.1145/1375581.1375595.

[5] F. Catthoor et al. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Boston, 1998. ISBN 978-1-4419-5061-1. doi:10.1007/978-1-4757-2849-1.

[6] P. Clauss, B. Kenmei, and J. C. Beyler. The periodic-linear model of program behavior capture. In *Proceedings of the 11th International Euro-Par Conference*, pages 325–335, Lisbon, Portugal, 2005. doi:10.1007/11549468_38.

[7] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Proceedings of the 10th International Euro-Par Conference*, pages 292–303, Pisa, Italy, 2004. doi:10.1007/978-3-540-27866-5_38.

[8] M. Diener, E. H. M. da Cruz, P. O. A. Navaux, A. Busse, and H. U. Heiß. kMAF: Automatic kernel-level management of thread and data affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques, PACT*, pages 277–288, Edmonton, AB, Canada, 2014. doi:10.1145/2628071.2628085.

[9] G. Gupta and S. Rajopadhye. The Z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 237–248, San Jose, CA, USA, 2007. doi:10.1145/1229428.1229478.

[10] J. Holewinski et al. Dynamic trace-based analysis of vectorization potential of applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 371–382, Beijing, China, 2012. doi:10.1145/2254064.2254108.

[11] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th Annual International Conference on Supercomputing, ICS*, pages 1–11, Saint Malo, France, 2004. doi:10.1145/1006209.1006211.

[12] I. Issenin and N. Dutt. FORAY-GEN: Automatic generation of affine functions for memory optimizations. In *Proceedings of the 2005 Design, Automation and Test in Europe Conference and Exposition, DATE*, pages 808–813, Munich, Germany, 2005. doi:10.1109/DATE.2005.157.

[13] A. Jimborean, P. Clauss, J. M. Martínez, and A. Sukumaran-Rajam. Online dynamic dependence analysis for speculative polyhedral parallelization. In *Proceedings of the 19th International Euro-Par Conference*, pages 191–202, Aachen, Germany, 2013. doi:10.1007/978-3-642-40047-6_21.

[14] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14 (3):563–590, 1967. doi:10.1145/321406.321418.

[15] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th International Symposium on Code Generation and Optimization, CGO*, pages 94–103, Boston, MA, USA, 2008. doi:10.1145/1356058.1356071.

[16] M. Kobayashi. Dynamic characteristics of loops. *IEEE Trans. Comput.*, 33(2):125–132, 1984. doi:10.1109/TC.1984.1676404.

[17] L. Lamport. The parallel execution of DO loops. *Commun. ACM*, 17(2):83–93, 1974. doi:10.1145/360827.360844.

[18] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static dissassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS*, pages 290–299, Washington, DC, USA, 2003. doi:10.1145/948109.948149.

[19] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, PLDI*, pages 190–200, Chicago, IL, USA, 2005. doi:10.1145/1065010.1065034.

[20] C. Mendis et al. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to Halide DSL code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 391–402, Portland, OR, USA, 2015. doi:10.1145/2737924.2737974.

[21] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th International Conference on Computing Frontiers, CF*, pages 143–152, Ischia, Italy, 2007. doi:10.1145/1242531.1242554.

[22] G. Piccoli et al. Compiler support for selective page migration in NUMA architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques, PACT*, pages 369–380, Edmonton, AB, Canada, 2014. doi:10.1145/2628071.2628077.

[23] L.-N. Pouchet. PolyBench: The polyhedral benchmark suite, 2011. URL `http://www.cs.ucla.edu/~pouchet/software/polybench/`. Last accessed: January 2016.

[24] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32:174–199, 2000. doi:10.1145/358923.358939.

[25] P. Yedlapalli et al. Meeting midway: Improving CMP performance with memory-side prefetching. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT*, pages 289–298, Edinburgh, United Kingdom, 2013. doi:10.1109/PACT.2013.6618825.