Reconstructing affine codes from their memory traces

Gabriel Rodríguez José M. Andión
Juan Touriño
University of A Coruña, Spain

Mahmut T. Kandemir
Pennsylvania State University, USA

Abstract

Complete comprehension of loop codes is desirable for a variety of program optimizations. Compilers perform static transforms and analyses, 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 code of target applications is 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 trace. 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, this achieves very high compression rates of trace files. When used for dynamic access characterization, it is capable of predicting over 99% of future memory accesses.

1. Introduction

Affine codes represent an important class of problems 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 variable values through affine bounds and subscripts, accessing and operating on large arrays of data. This is the type of codes that is usually modeled and optimized using the polyhedral approach [4, 9, 13, 16].

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, programmers may use complex data and control structures, including code obfuscation techniques, that mask the underlying application logic and prevent static analysis and optimization.

This paper presents an exploratory approach for *automatically reconstructing* affine loops from a trace of their memory accesses. The exploration engine traverses a tree-like space. Conceptually, level k in this tree 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 variables, and only adds a new level when no other solution is feasible. The basic approach explores the entire solution space for the trace 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 extract the reconstructed 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, if it exists, and 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 it is not guaranteed that the number of levels in the reconstructed loop will be the same as in the original code. In particular, it is possible that the reconstructed code has fewer levels. A mechanism for increasing the final number of levels in a user-guided way is also proposed. The main **contributions** of this work are:

- A mathematical framework for the extraction of an affine representation of a given memory trace (Section 2). Stragies 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. (Section 3). These results show that: (i) the system 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 instruction.

Potential applications of the framework include trace compression, approximation of nearly affine traces, analysis of codes with complex control or obfuscated codes, as well as online optimizations such as dependence detection for automatic parallelization and memory prefetching. These are discussed in depth in Section 4, along with the related work.

2. Trace-based reconstruction

2.1 Mathematical formulation

The proposed reconstruction algorithm assumes that the trace contains, at least, the memory address of the instruction issuing the access, or a similar way to uniquely identify the instruction, and the accessed location. In the general case, it is expected that a trace file will contain the entire execution of the program, including multiple loop nests and non-loop sections. Detection of loop sections in execution traces falls out of the scope of this paper, but has been discussed in previous work [15, 19]. In this paper a reliable mechanism to detect and extract loop sections in the trace is assumed. It is also assumed that each source instruction accesses a single data ar-

ray in the code, and that the loops bounds and subscripts are affine. Without loss of generality, these types of loops can be written as:

$$\begin{array}{l}\texttt{DO } i_1 = 0,\ u_1(\overrightarrow{i}) \\ \quad \texttt{DO } i_2 = 0,\ u_2(\overrightarrow{i}) \\ \qquad \vdots \\ \quad \texttt{DO } i_n = 0,\ u_n(\overrightarrow{i}) \\ \qquad V[f_1(\overrightarrow{i})]\ldots[f_m(\overrightarrow{i})]\end{array}$$

where $\{u_j, 0 < j \leq n\}$ are affine functions that provide the upper bounds of loop i_j; $\{f_d(i_1, \ldots, i_n), 0 < d \leq m\}$ 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 $\overrightarrow{i} = \{i_1, \ldots, i_n\}^T$ is a column vector which encodes the state of each iteration variable. The particular set of index values for the k^{th} execution of access V is denoted by $\overrightarrow{i}^k = \{i_1^k, \ldots, i_n^k\}^T$; and the complete access $V[f_1(\overrightarrow{i})]\ldots[f_m(\overrightarrow{i})]$ is abbreviated by $V(\overrightarrow{i})$. Note that each upper bounds function $u_j(\overrightarrow{i})$ can only depend on scoped variables at the nesting level j, i.e., $\{i_1, \ldots, i_{j-1}\}$. This is not explicitly acknowledged to simplify notation. Iteration bounds are assumed to be inclusive, i.e., $0 \leq i_j \leq u_j(\overrightarrow{i})$. Since f_j is affine, the access can be rewritten as:

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

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.

During the execution of the loop nest, the instruction which implements the access to V will orderly issue the addresses corresponding to $V(\overrightarrow{i}^1)$, $V(\overrightarrow{i}^2)$, $V(\overrightarrow{i}^3)$, etc. These addresses will be registered in the trace file together with the instruction issuing them and the size of the accessed data. This memory trace format can be generated, for instance, by Intel Pin [18].

Consider two consecutive accesses, $V(\overrightarrow{i}^k)$ and $V(\overrightarrow{i}^{k+1})$, and assume that the loop index values in $\overrightarrow{i}^k = \{i_1^k, \ldots, i_n^k\}$ and the upper bounds functions, $u_1(\overrightarrow{i}), \ldots, u_n(\overrightarrow{i})$ are known. The values in $\overrightarrow{i}^{k+1}$ can be calculated as follows:

1. An index i_j will be reset to 0 if, and only if all of the following hold:
   - All innermore indices are resetting.
   - Either i_j has reached its maximum iteration count, or some innermore index has a negative value for its maximum iteration count when i_j increases by one:

   $$(i_j = u_j(\overrightarrow{i}^k)) \vee \left(\exists l, j < l \leq n; u_l(\ldots, i_j^k + 1, \ldots) < 0\right)$$

2. An index i_j will be increased by one if, and only if all of the following hold:
   - All innermore indices are resetting.
   - i_j has not reached its maximum iteration count, and all innermore indices have non-negative values for their maximum iteration count when i_j increases by one:

   $$(i_j < u_j(\overrightarrow{i}^k)) \wedge \left(\forall l, j < l \leq n; u_l(\ldots, i_j^k + 1, \ldots) \geq 0\right)$$

3. In any other case, i_j will not change.

[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).

These conditions are intuitive and a direct consequence of loop semantics and application control flow. If any internal index $(i_l, j < l \leq n)$ is not resetting, then control flow will not exit the loop at level l, and therefore it will be impossible for i_j to be modified. If all internal indices reset, then control flow will reach the post-loop section of loop at level j, increasing i_j by one unit. If $i_j^k = u_j(\overrightarrow{i}^k)$ then this increase will cause the index to be beyond its maximum iteration count, and control flow will exit level j. If there is an iteration $(k + 1)$, then control flow must re-enter level j later, executing the pre-loop instruction and assigning $i_j = 0$. If $i_j^k < u_j(\overrightarrow{i}^k)$ but there is some inner level l such that its maximum iteration count takes a negative value when i_j is increased by one unit, then control flow will not enter level l, will not reach V, and no memory access may be executed until i_j resets to 0. In any other case, the next access to V will be performed in iteration $\overrightarrow{i}^{k+1} = \{i_1^k \ldots, i_j^k + 1, 0, \ldots, 0\}$.

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

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 $\Rightarrow \delta_j^k = 0$
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:

$$(\overrightarrow{i}^{k+1} - \overrightarrow{i}^k) = \begin{bmatrix} i_1^{k+1} - i_1^k \\ i_2^{k+1} - i_2^k \\ \vdots \\ i_n^{k+1} - i_n^k \end{bmatrix} = \begin{bmatrix} \delta_1^k \\ \delta_2^k \\ \vdots \\ \delta_n^k \end{bmatrix} = \overrightarrow{\delta}^k$$

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:

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

□

2.2 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 (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. 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 Figure 1. Starting from the k^{th} iteration vector $\overrightarrow{i}^k = \{i_1^k, \ldots, i_n^k\}$ there are $(2n + 1)$ different vectors $\overrightarrow{i}^{k+1}$ that are considered as candidates for the $(k + 1)^{th}$ iteration vector. If a solution exists, the algorithm builds the minimal nest capable of generating the observed access trace[2]. This section models the

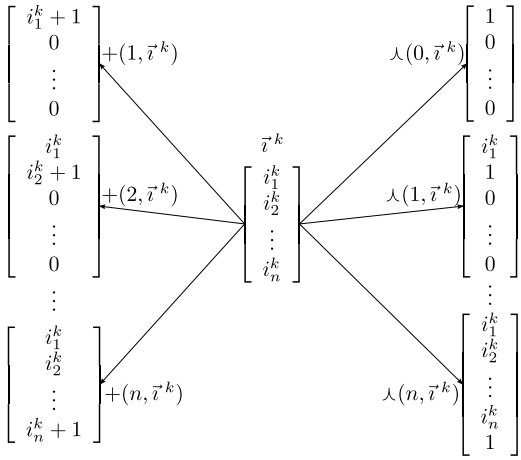

Figure 1. Solution space. For each reconstructed index $\overrightarrow{i}^k$ there are $(2n+1)$ possible values for $\overrightarrow{i}^{k+1}$. The n alternatives on the left side are obtained using an operation $+(j, \overrightarrow{i})$ that increases index i_j by one, and resets to zero all innermore indices. The $(n+1)$ alternatives on the right are obtained by applying an operation $\curlywedge(j, \overrightarrow{i})$, which inserts a new loop at nesting level $(j+1)$.

problem, and develops exploration strategies to efficiently traverse the solution space taking into account its mathematical characteristics.

Let $\overrightarrow{a} = \{a_1, \ldots, a_N\} = \{V(\overrightarrow{i}^1), \ldots, V(\overrightarrow{i}^N)\}$ be the addresses accessed by a single instruction, included in the execution trace. Since the upper bounds functions are affine, each $u_j(\overrightarrow{i})$ can be written as:

$$u_j(\overrightarrow{i}) = w_j + u_{j,1} i_1 + \ldots + u_{j,(j-1)} i_{(j-1)} \tag{2}$$

and therefore it is possible to build a matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$ and a column vector $\overrightarrow{w} \in \mathbb{Z}^n$ such that:

$$\mathbf{U} = \begin{bmatrix} -1 & 0 & 0 & \ldots & 0 \\ u_{2,1} & -1 & 0 & \ldots & 0 \\ u_{3,1} & u_{3,2} & -1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_{n,1} & u_{n,2} & u_{n,3} & \ldots & -1 \end{bmatrix} \quad \overrightarrow{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \tag{3}$$

Note that $\mathbf{U}$ is a lower triangular matrix, since no index i_j can depend on an innermore index; and that its main diagonal is equal to $\overrightarrow{-1} \in \mathbb{Z}^n$. Using $\mathbf{U}$ and $\overrightarrow{w}$, the condition for a given iteration tuple $\overrightarrow{i}$ to be valid under the loop constraints in the canonical loop form can be written as:

$$\mathbf{U}\overrightarrow{i} + \overrightarrow{w} \geq \overrightarrow{0}^T \tag{4}$$

Let us assume that the algorithm has already identified a partial solution $\mathcal{S}_n^k = \{\overrightarrow{c}, \mathbf{I}^k, \mathbf{U}, \overrightarrow{w}\}$, which reconstructs the subtrace $\{a_1, \ldots, a_k\}$ using n nested loops, whose components are defined as follows:

- Vector $\overrightarrow{c} \in \mathbb{Z}^n$ of coefficients of loop indices.
- Matrix $\mathbf{I}^k = [\overrightarrow{i}^1 | \ldots | \overrightarrow{i}^k] \in \mathbb{Z}^{n \times k}$ of reconstructed indices.
- Matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$, bounds matrix as defined in Eq. (3).
- Vector $\overrightarrow{w} \in \mathbb{Z}^n$, bounds vector as defined in Eq. (3).

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

To be a valid solution, $\mathcal{S}_n^k$ has to meet the following requirements:

1. Each consecutive pair of indices $\overrightarrow{i}^k$ and $\overrightarrow{i}^{k+1}$ must be sequential as per Definition 2.1.

   Note that this condition is stronger than simply requiring that the iteration indices stay inside the loop bounds, which could be written extending Eq. (4) as:

   $$\mathbf{U}\mathbf{I}^k + \overrightarrow{w}\mathbf{1}^{1 \times k} \geq \mathbf{0}^{n \times k} \tag{5}$$

2. The observed strides are coherent with the reconstructed ones. Using Lemma 2.2 this is written as:

   $$\overrightarrow{c}(\overrightarrow{i}^{k+1} - \overrightarrow{i}^k) = \overrightarrow{c}\,\overrightarrow{\delta}^k = \sigma^k$$

Upon processing access a_{k+1}, the algorithm first calculates the observed access stride:

$$\sigma^k = a_{k+1} - a_k \tag{6}$$

Afterwards, it builds a diophantine[3] linear equation system based on Lemma 2.2 to discover the potential indices $\overrightarrow{i}^{k+1}$ which generate an access stride equal to the observed one:

$$\overrightarrow{c}(\overrightarrow{i}^{k+1} - \overrightarrow{i}^k) = \sigma^k \Rightarrow (\overrightarrow{c}^T\overrightarrow{c})\,\overrightarrow{\delta}^k = \overrightarrow{c}^T\sigma^k \tag{7}$$

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 $\overrightarrow{i}^{k+1} = \overrightarrow{i}^k + \overrightarrow{\delta}^k$ is calculated, and $\mathbf{I}^{k+1} = \left[\mathbf{I}^k | \overrightarrow{i}^{k+1}\right]$. $\mathbf{U}$, $\overrightarrow{w}$, and $\overrightarrow{c}$ remain unchanged. Each of these solutions must be explored independently.
2. The system has no solution, in which case there are three courses of action:
   (a) Modify the boundary conditions imposed by $\mathbf{U}$ and $\overrightarrow{w}$.
   (b) Increase the dimensionality of the solution: compute $\mathcal{S}_{n+1}^{k+1}$ modeling a loop with $(n+1)$ nesting levels.
   (c) Discard this branch.

Section 2.3 describes heuristic methods to guide the search through the solution space to accelerate traversal.

2.2.1 Solving the linear diophantine system

Although the system in Eq. (7) 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 ad hoc solving strategies.

Lemma 2.3. *There are at most n valid solutions to the system in Eq. (7). These correspond to indices:*

$$\{\overrightarrow{i}_l^{k+1} = +(l, \overrightarrow{i}^k), 0 < l \leq n\}$$

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

[3] The system must be diophantine, as loop indices may only have integer values.

the position δ_l^k that is equal to 1.

$$\begin{bmatrix} \delta_1^k \\ \vdots \\ \delta_{l-1}^k \\ \delta_l^k \\ \delta_{l+1}^k \\ \vdots \\ \delta_n^k \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ -i_{l+1}^k \\ \vdots \\ -i_n^k \end{bmatrix} \tag{8}$$

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 indices $\overrightarrow{\imath}_l^{k+1}$, calculating the predicted stride for each combination as $\hat{\sigma}_l^k = \overrightarrow{c}\,\overrightarrow{\delta}_l^k$, and accepting those solutions that generate a stride equal to the observed one, $\hat{\sigma}_l^k = \sigma^k$, obtained using Eq. (6). These will be particular solutions of the subtrace $\{a_1, \ldots, a_{k+1}\}$, which can be explored to construct a solution for the entire trace.

2.3 Exploration of the solution space

2.3.1 Branch priority

The proposed approach is capable of efficiently finding the relevant solutions of the linear diophantine system for each address of the trace, but will 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 $O(n^N)$. 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:

$$\overrightarrow{\gamma}^k = \mathbf{U}\overrightarrow{\imath}^k + \overrightarrow{w} \tag{9}$$

Lemma 2.4. *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, defined in Eq. (2), minus the current value of i_j:

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

$$= u_j(\overrightarrow{\imath}) - i_j$$

where $\mathbf{U}_{(j,:)}$ denotes the j^{th} row of matrix $\mathbf{U}$. 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 $\overrightarrow{\imath}_l^{k+1} = +(l, \overrightarrow{\imath}^k)$, where l is the position of the rightmost positive element of $\overrightarrow{\gamma}^k$.

The correctness of $\overrightarrow{\imath}_l^{k+1}$ 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. (4), which further improves the efficiency of the approach by saving calculations.

2.3.2 Extracting loop bounds

So far it has been assumed that the boundary conditions, $\mathbf{U}$ and $\overrightarrow{w}$, can be used to correctly predict $\overrightarrow{\imath}^{k+1}$ from $\overrightarrow{\imath}^k$. This is not true in the general case, as initially the loop bounds are unknown, as are the number of loops involved in the execution of the instruction accessing V.

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 rightmost positive element of $\overrightarrow{\gamma}^k$, as described before. 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 loop index $i_{l'}$ will have to be selected as described in Section 2.2.1, but the constructed $\mathbf{I}^{k+1}$ will not be valid in the context of the extracted loop bounds, $\mathbf{U}$ and $\overrightarrow{w}$, because either $\overrightarrow{\imath}_{l'}^{k+1}$ will not be sequential to $\overrightarrow{\imath}^k$, or it will violate boundary conditions. In this situation it is necessary to generate new boundary conditions $\mathbf{U}'$ and $\overrightarrow{w}'$. These can be found by solving the system in Eq. (5):

$$\mathbf{U}'\mathbf{I}^{k+1} + \overrightarrow{w}'\mathbf{1}^{1\times(k+1)} \geq \mathbf{0}^{n\times(k+1)} \tag{10}$$

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 outer variables cannot be affected by inner, unscoped ones. As such, their first $(l-1)$ rows are known. Besides, in order for the indices to be sequential, it is necessary to build $\mathbf{U}'$ and $\overrightarrow{w}'$ such that loop resets as predicted by $\overrightarrow{\gamma}$ are consistent with loop resets observed in $\mathbf{I}^{k+1}$.

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 $\{w'_j, l \leq j \leq n\} \in \overrightarrow{w}'$, consider the reduced system:

$$\mathbf{U'}_{(j,:)}\overrightarrow{\imath}^z + w'_j \geq 0 \Rightarrow \sum_{r=1}^{j} u'_{j,r}i_r^z + w'_j \geq 0 \tag{11}$$

where $\mathbf{U'}_{(j,:)}$ is currently unknown, and $\overrightarrow{\imath}^z \in \mathbf{I}^{k+1}$ is arbitrarily selected. In order to calculate w'_j it is possible to take advantage of the properties of the canonical loop form, by choosing an $\overrightarrow{\imath}^z$ such that:

$$\overrightarrow{\imath}^z = \left[0, \ldots, 0, i_j^z, \ldots, i_n^z\right]^T$$

Since every loop index must start at 0 and by the sequential construction of the columns of $\mathbf{I}^{k+1}$, such an i^z is guaranteed to exist. Replacing it in the previous equation and taking into account that the main diagonal of $\mathbf{U}'$ must be equal to $\overrightarrow{-1} \in \mathbb{Z}^n$:

$$\sum_{r=1}^{j} u'_{j,r}i_r^z + w'_j \geq 0 \Rightarrow u'_{j,j}i_j^z + w'_j \geq 0 \Rightarrow w'_j \geq i_j^z$$

Lemma 2.5. *In order to guarantee that the bounds conditions in Eq. (5) hold, $\overrightarrow{\imath}^z$ must be chosen out of all the possible candidates such that i_j^z is maximum, and w'_j must be equal to i_j^z.*

Proof. If w'_j was not selected to be equal to some i_j^z, then $\mathbf{I}^{k+1}$ would not be sequential as per Definition 2.1 under the boundary conditions established by $\overrightarrow{w}'$. Now, assume that an $\overrightarrow{\imath}^{z'}$ is selected such that $i_j^{z'}$ is not maximum, $i_j^z > i_j^{z'}$. Then:

$$\mathbf{U}_{(j,:)}\overrightarrow{\imath}^z + w'_j = -i_j^z + i_j^{z'} < 0$$

The constructed $\overrightarrow{w}'$ would not be consistent with some of the entries in $\mathbf{I}^{k+1}$. □

Corollary 2.6. *In Eq. (11), it is only necessary to calculate the value w'_l, as other elements of $\overrightarrow{w}$ will remain unchanged. Moreover,*

w'_l will only change if $(\forall j, 0 < j < l, i_j^{k+1} = 0)$ *and, in that case,* $w'_l = i_l^{k+1}$.

$$\overrightarrow{w}' = \left[w_1, \ldots, w_{l-1}, \boldsymbol{w'_l}, w_{l+1}, \ldots, w_l\right]^T$$

Proof. If $\{w'_j, l < j \leq n\}$ can be calculated exclusively selecting vectors in the shape of $\overrightarrow{\imath}^z$ as per Lemma 2.5, then index $\overrightarrow{\imath}_l^{k+1} = +(l, \overrightarrow{\imath}^k)$ is not a feasible selection for $\overrightarrow{\imath}^z$ when calculating w'_j, since $i_l^{k+1} > 0$ by definition of the $+$ operation. Therefore, w'_j will be equal to the one calculated for the previous step of the algorithm using $\mathbf{I}^k$. Using the same reasoning, if $(\exists j, 0 < j < l, i_j^{k+1} \neq 0)$, index $\overrightarrow{\imath}_l^{k+1}$ is not a feasible selection for calculating w'_l. Otherwise, and by definition of the $+$ operation and index sequentiality, $i_l^{k+1} = i_l^z$ will be maximum. □

Using this result, the calculation of $\overrightarrow{w}'$ becomes $O(1)$. Once $\overrightarrow{w}'$ is calculated, the unknown rows $\{\mathbf{U}'_{(j,:)}, l \leq j \leq n\}$ can be calculated by reducing the original system in Eq. (10) to $(n-l+1)$ equation systems of the form:

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

where $\mathbf{i}^z \in \mathbb{Z}^{n\times n}$ is a full rank matrix of columns extracted from $\mathbf{I}^{k+1}$. As established in Lemma 2.5, it is necessary to choose $\mathbf{i}^z = \{\overrightarrow{\imath}_1^z, \ldots, \overrightarrow{\imath}_n^z\}$ such that each of its columns represents an iteration where index i_j is maximum for a specific combination of indices $(i_0, \ldots, i_{j-1})$. Note that the inequality in Eq. (10) has been changed to ensure that $\gamma_j = u_j(\overrightarrow{\imath}) = 0$ will hold for each of the selected iterations, guaranteeing index consistency.

In order to efficiently solve these systems, two optimizations can be considered. First, since $\mathbf{U}'$ must be a lower triangular matrix with known main diagonal, the previous system can be reduced to:

$$\mathbf{U}'_{(j,1:j)}\mathbf{i}^z_{(1:j,1:j-1)} + w'_j\mathbf{1}^{1\times(j-1)} = \mathbf{0}^{1\times(j-1)} \tag{12}$$

where $\mathbf{U}'_{(j,1:j)} \in \mathbb{Z}^{1\times j}$ denotes the first j entries of the j^{th} row of $\mathbf{U}'$, and $\mathbf{i}^z_{(1:j,1:j-1)} \in \mathbb{Z}^{j\times(j-1)}$ denotes the first $(j-1)$ entries in the first j rows of matrix $\mathbf{i}^z$. Only $(j-1)$ indexes are needed, as that is the number of unknowns in the j^{th} row of $\mathbf{U}'$. Second, note that any full rank matrix can be extracted from $\mathbf{I}^{k+1}$ to build $\mathbf{i}^z$ as long as the selected columns are iterations where index i_j is maximum. By taking advantage of the canonical loop form, this means that it is always possible to build $\mathbf{i}^z$ as a triangular matrix, and solve the system in linear time $O(j)$. By applying both optimizations the calculation of $\mathbf{U}'$ becomes $O(n^2)$.

2.3.3 Extracting the coefficients of loop indices

Once again, assume that the algorithm has found a partial solution $\mathcal{S}_n^k = \{\overrightarrow{c}, \mathbf{I}^k, \mathbf{U}, \overrightarrow{w}\}$. If no valid $\{\overrightarrow{\imath}_l^{k+1} = +(l, \overrightarrow{\imath}^k), 0 < l \leq n\}$ can be built using the methods described in Sections 2.3.1 and 2.3.2 it may be caused by a loop index increasing in access $(k+1)$ which had not appeared before. This can cause σ^k to be unrepresentable either as a linear combination of the currently known coefficients in $\overrightarrow{c}$, or as a set of sequential indices $\mathbf{I}^{k+1}$. Assuming that the first k accesses have been correctly recognized, 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 $(n+1)$ potential solutions that need to be explored, as shown in the right half of Figure 1, one for each insertion position of the newly discovered index. The most common situation, particularly for large values of k, is that newly discovered loops are outer than the previously known ones. In any case, given an insertion point $(p, 0 \leq p \leq n)$ for the new loop index i_p, the set of indices $\mathbf{I}^{k+1} \in \mathbb{Z}^{(n+1)\times(k+1)}$ is generated as follows:

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

where a 0 in position p has been added to each index $\overrightarrow{\imath} \in \mathbf{I}^k$, and a new column $\overrightarrow{\imath}^{k+1} = \curlywedge(p, \overrightarrow{\imath}^k)$ has been added to the matrix. The coefficient c'_p associated to the new loop index can be derived from Eq. (7):

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

$$\left[c_1, \ldots, c_p, \boldsymbol{c'_p}, c_{p+1}, \ldots, c_n\right]\left[\begin{array}{c} 0 \\ \vdots \\ 0 \\ 1 \\ -i_p^k \\ \vdots \\ -i_n^k \end{array}\right] = \sigma^k \Rightarrow$$

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

After calculating the new $\overrightarrow{c}$, $\mathbf{U}$ and $\overrightarrow{w}$ are updated as described in Section 2.3.2 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 $\{a_1, \ldots, a_N\}$ can be generated using at most N affine nested loops. For this reason, the solution space should be traversed in a breadth-first fashion, to ensure that a minimal solution, in terms of number of generated nested loops, is reached.

2.3.4 Starting the exploration

In the previous sections it has been discussed how to constructively build a solution for the subtrace $\{a_1, \ldots, a_{k+1}\}$ assuming that the solution for $\{a_1, \ldots, a_k\}$ is known. The first partial solution $\mathcal{S}_1^2$ for $\{a_1, a_2\}$ is built as:

- $\overrightarrow{c} = [\sigma^1]$
- $\mathbf{I}^2 = [\overrightarrow{\imath}^1 | \overrightarrow{\imath}^2] = [0, 1]$
- $\mathbf{U} = [-1]$
- $\overrightarrow{w} = [1]$

It can be proven that this represents the only feasible solution for the subtrace $\{a_1, a_2\}$. The exploration engine can then begin working, gradually increasing the size of the partial solution, until it reaches a solution for the entire trace $\overrightarrow{a}$, or it discards the root of the solution space, $\mathcal{S}_1^2$, in which case $\overrightarrow{a}$ cannot be generated by an affine loop.

2.4 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 function that calculates new loop insertions described in Section 2.3.3 has been encapsulated into a `Grow()` function, shown in Algorithm 2. The extraction starts by calling `Extract()` with the initial $\mathcal{S}_1^2$ defined in the previous section. In the worst case, when no access can be predicted using $\overrightarrow{\gamma}$, the algorithm uses the brute force approach $(O(n^N))$. In the best case every access is predicted by $\overrightarrow{\gamma}$ $(O(N))$.

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.

Algorithm 1: Pseudocode of `Extract()`

Input: the execution trace, $\overrightarrow{a}$, 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 $k < len(\overrightarrow{a}) - 1$ **do**
  $\sigma = a_{k+1} - a_k$;
  `// Try to use` $\overrightarrow{\gamma}$ `(§2.3.1)`
  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 (§2.2.1)`
  for x=n *down to* 1 **do**
    calculate $\hat{\sigma}_x = \overrightarrow{c}\,\overrightarrow{\delta}_x$;
    if $\hat{\sigma}_x = \sigma$ **then**
      $\mathbf{I}' = [\mathbf{I}| + (x, \overrightarrow{\imath}^k)]$;
      $\{\mathbf{U}', \overrightarrow{w}'\}$ = update bounds; `// §2.3.2`
      if $\{\overrightarrow{c}, \mathbf{I}', \mathbf{U}', \overrightarrow{w}'\}$ *is linear* **then**
        $\mathcal{S}' = Extract(\{\overrightarrow{c}, \mathbf{I}', \mathbf{U}', \overrightarrow{w}'\}, \overrightarrow{a})$;
        if $\mathcal{S}' \neq None$ **then** return $\mathcal{S}'$;
      end
    end
  end
  `// Add loop (§2.3.3)`
  for x=0 *to* n **do**
    $\mathcal{S}' = Extract(Grow(\mathcal{S}, x), \overrightarrow{a})$;
    if $\mathcal{S}' \neq None$ **then** return $\mathcal{S}'$;
  end
  return $None$;
end

Algorithm 2: Pseudocode of `Grow()` (§2.3.3)

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

3. Experimental Evaluation

The proposed algorithm has been implemented in Python and used to extract codes for different affine kernels. Each execution was performed on an Intel Xeon E5-2660 Sandy Bridge 2.20 Ghz node, with 64 GB of RAM. The reconstruction algorithm was run with traces generated by the PolyBench/C 3.2 suite [20]. It includes 30 applications from domains such as linear algebra, stencil codes and data mining, out of which 28 were selected. The target was the traces generated by the kernels[4] of these applications. 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` (150 MB in disk) to 12.9 billion references for `3mm` (270 GB).

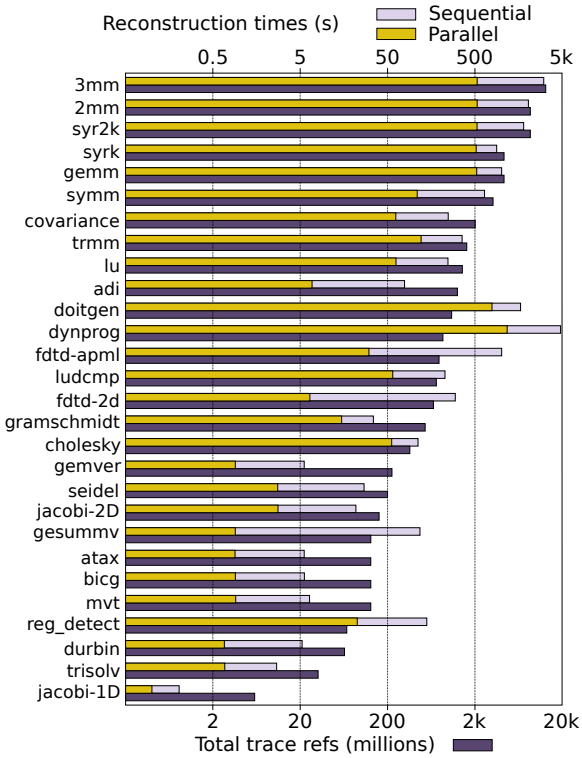

Figure 2. Reconstruction times (upper axis) and trace sizes (lower axis) for the PolyBench/C benchmarks, ordered by trace size. Axes are logarithmic. Since the subtraces of a kernel are independent they can be reconstructed in parallel, achieving an average speedup of 5.6x.

Figure 2 shows trace sizes and processing times. These largely depend on the number of reconstructed loops, as well as on which of the loops iterate the most. 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 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 180.000

[4] These are the parts marked with `scop` pragmas in PolyBench/C and represent the vast majority of the accesses issued by the entire program.

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

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

Trace	%	Trace	%	Trace	%	Trace	%
3mm	99.85	trmm	99.97	fdtd-2d	98.00	atax	74.96
2mm	99.84	lu	99.71	grams.	99.61	bicg	74.96
syr2k	99.85	adi	98.00	chol.	99.99	mvt	87.46
syrk	99.83	doit.	98.83	gemv.	78.53	reg_d.	99.78
gemm	99.83	dynp.	99.98	seidel	95.00	durbin	99.88
symm	99.80	fdtd-a.	75.62	jac-2D	95.00	trisolv	99.89
covar.	99.70	lud.	99.99	gesum.	74.95	jac-1D	99.00

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

references/second, in the fastest one this goes up to 30 million (167x faster).

A second set of experiments was run deactivating $\overrightarrow{\gamma}$ prediction. The engine must explore all potentially correct branches as indicated in Section 2.2.1. All subtraces were processed in parallel. The recognition was run for 48 hours, at which point 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 1 million references.

The usability of the engine as an online predictor was also evaluated. Table 2 shows the percentage of predicted accesses. 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, these references are trivially predicted by single-stride prefetching techniques [21]. A simple heuristic for predicting this type of references is to consider that when $\overrightarrow{\gamma}$ is not yet operational, by default the outermost discovered loop will iterate. The use of this heuristic increases prediction rate above 99% for all codes.

Regarding memory requirements, the exploration engine needs to store, at least, $\overrightarrow{c}$, $\overrightarrow{w}$, $\mathbf{U}$, and selected indices of $\mathbf{I}$[5]. In addition to these, some memory is consumed by backtracking points used to efficiently implement the recursivity in Algorithm 1. Total memory requirements for subtraces in our experimental set-up vary between 48 bytes and 60 KB.

4. Related Work and Applications

Not many works, to the best of the authors' knowledge, have explored the reconstruction of loop codes from 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 potential applications of the proposed exploration engine.

[8] 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. [7] introduced polyhedra to graphically represent the program memory behavior (including cache misses) and facilitate its understanding. [14] 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 a triplet of terms that can be rewritten as a loop.

On the topic of trace compression, [17] proposed whole program paths (WPP), a compressed directed acyclic path trace format. [22] developed a compaction technique for WPPs by eliminating redundant path traces and organizing trace information according to a dynamic call graph. [5] proposed VPC, a family of four compression algorithms that employ value predictors to compress extended program traces. These include PCs, contents of registers, or values on a bus.

One potential use of the exploration engine is cache prefetching. To improve on the one block lookahead scheme [21], [2] use a prediction table and lookahead program counter to preload regular accesses which correctly predict the stride of the innermost loop. [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 loop reconstruction requires memories 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 size of these memories depends on the maximum nesting level supported.

Trace-based code reconstruction is also useful for automatic parallelization. [10] use dynamic data dependence graphs derived from sequential execution traces to identify vectorization opportunities. [12] proposed a dynamic mechanism for detecting data dependences using interpolated linear functions to approximate observed memory accesses to guide speculative parallelization. Similar systems can be constructed using the proposed exploration engine, capable of analyzing dependences without the need for compiler support.

Some authors have approached the problem of designing ad hoc memory hierarchies for embedded applications. [6] proposed a compiler-based methodology to derive optimal memory regions and associated data allocation. [1] use a trace-based method that analyzes the access histogram to determine which memory regions to allocate to scratchpad memory [3]. Our trace-based reconstruction approach can be employed to design custom memory hierarchies without access to the source code. This is particularly interesting for IP cores, commonly included in embedded devices. It can also be employed to drive scratchpad allocation managers.

5. Concluding Remarks

This work has explored the reconstruction of affine loop codes from their memory traces, focusing on one instruction at a time. This problem has applications in trace compression, memory management and design, dynamic parallelization, or program analysis. 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

[5] These are used when recalculating $\overrightarrow{w}$ and $\mathbf{U}$ (see §2.3.2). It is not necessary to store the entire matrix $\mathbf{I}$ if memory requirements are to be optimized.

have been proposed. Experimental evaluation has shown good performance and accuracy in reconstructing affine codes, and significant overheads when processing nonlinearities. Furthermore, it has been shown that the problem is not tractable without the proposed optimizations.

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] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The VPC trace-compression algorithms. *IEEE Trans. Comput.*, 54(11):1329–1344, 2005. doi: 10.1109/TC.2005.186.

[6] F. Catthoor, S. Wuytack, G. E. de Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory management methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Boston, 1998. ISBN 978-1-4419-5061-1.

[7] P. Clauss and B. Kenmei. Polyhedral modeling and analysis of memory access profiles. In *Proceedings of the 2006 IEEE International Conference on Application-Specific Systems, Architecture and Processors, ASAP*, pages 191–198, Steamboat Springs, CO, USA, 2006. doi: 10.1109/ASAP.2006.54.

[8] 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.

[9] 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.

[10] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *Proceedings of the 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] 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.

[13] 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.

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

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

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

[17] J. R. Larus. Whole program paths. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 259–269, Atlanta, GA, USA, 1999. doi: 10.1145/301618.301678.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 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.

[19] 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.

[20] L.-N. Pouchet. PolyBench: The polyhedral benchmark suite. http://www.cs.ucla.edu/~pouchet/software/polybench/, 2011.

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

[22] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 180–190, Snowbird, UT, USA, 2001. doi: 10.1145/378795.378835.