A tool for reconstructing codes from memory traces

José M. Andión*, Gabriel Rodríguez*,
Mahmut T. Kandemir†,
Juan Touriño*,[1],[2]

* *Grupo de Arquitectura de Computadores, Departamento de Electrónica e Sistemas, Universidade da Coruña, 15071 A Coruña, Spain*
† *Microsystems Design Lab, Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA*

ABSTRACT

We propose a tool for rebuilding affine loop nests from a trace of memory accesses, without user intervention or usage of source/binary codes. The reconstruction has been formalized as the traversal of a tree-like solution space, in which each node symbolizes a point in the iteration space of a loop. Our tool efficiently traverses this space, being able to process hundreds of gigabytes of trace in minutes. Potential applications include hardware and software prefetching, data placement for locality optimization, dependence analysis for automatic parallelization, optimal design of embedded memory systems, and trace compression.

KEYWORDS: trace analysis; polyhedral optimization; program behavior modeling

1 Introduction

Many program optimizations focus on loops, as they often consume the biggest part of the execution time. Compilers based on the polyhedral model (e.g., Pluto [BHRS08] and Polly [GGL12]) have been very successful for this purpose, but the source code is not always available/adequate to build the program representation. Another example are intellectual property (IP) cores in embedded systems: they have a well-known high-level functionality, but their internals are opaque to the system designer and the programmer. Thus, it becomes necessary to find alternate ways to model application behavior. This manuscript summarizes the main ideas introduced in [RAKT16] for the automatic reconstruction of affine codes from a trace of their memory accesses.

[1]Contact: `http://gac.udc.es/~grodriguez/trace-reconstruction/`

[2]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; NSF grants 1213052, 1439021, 1409095, and 1526750; and a grant from Intel.

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 work, 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 Pin). Hence, the address stream generated by each instruction can be analyzed separately.

The algorithm focuses on the reconstruction of each individual reference enclosed in large regular loops, with linear static control parts that depend only on the loop index variables and loop independent constants through affine bounds and subscripts. These types of loops are the main target of the polyhedral model and can be written as:

```
DO $i_1$ = 0, $u_1(\overrightarrow{\imath})$
  ...
  DO $i_n$ = 0, $u_n(\overrightarrow{\imath})$
    $V[f_1(\overrightarrow{\imath})]\dots[f_m(\overrightarrow{\imath})]$
```

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 loop nest to a point in the data space of V, and $\overrightarrow{\imath}^k = \{i_1^k, \dots, i_n^k\}^T$ is a column vector which encodes the state of each iteration variable for the k^{th} execution of the loop nest. The complete access $V[f_1(\overrightarrow{\imath})]\dots[f_m(\overrightarrow{\imath})]$ is abbreviated by $V(\overrightarrow{\imath})$. Since f_d are affine, the access can be rewritten as:

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

where c_0 is a constant stride and each $\{c_j, 0 < j \leq n\}$ is the coefficient of the loop index i_j.

During the execution of the loop nest, the access to V will orderly issue the addresses corresponding to $V(\overrightarrow{\imath}^1)$, $V(\overrightarrow{\imath}^2)$, etc. Note that, using Eq. (1), the stride between two consecutive accesses $\sigma^k = V(\overrightarrow{\imath}^{k+1}) - V(\overrightarrow{\imath}^k)$ can be expressed as a linear combination of the coefficients of the loop indices. This is the basis of our reconstruction approach.

3 Reconstruction Method

The proposed technique is essentially a guided exploration of a tree-like potential solution space driven by the access strides, 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. Its root is a trivial loop that generates the first two accesses in the trace. The exploration engine incorporates 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. Starting from the k^{th} iteration vector $\overrightarrow{\imath}^k = \{i_1^k, \dots, i_n^k\}$ there are $(2n+1)$ different vectors $\overrightarrow{\imath}^{k+1}$ that are considered as candidates for the $(k+1)^{th}$ iteration vector. The n alternatives on the left side are obtained using an operation $+(j, \overrightarrow{\imath})$, which 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 $(j+1)$.

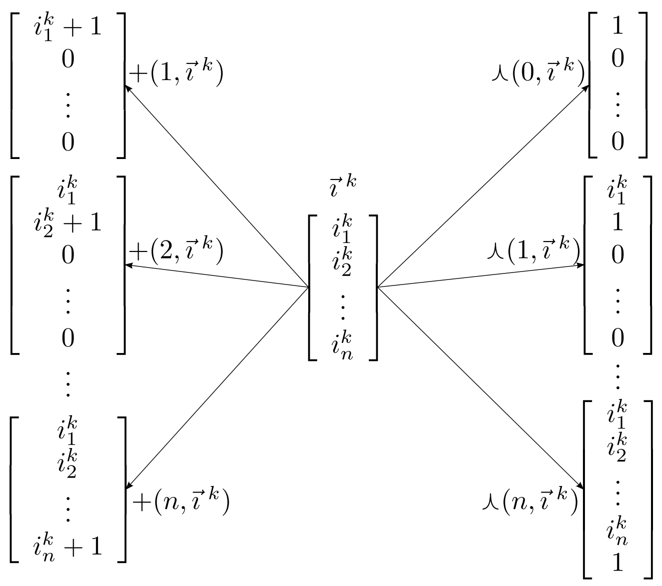

$\vec{\imath}^{\,k}$
$+(1,\vec{\imath}^{\,k})$
$+(2,\vec{\imath}^{\,k})$
$+(n,\vec{\imath}^{\,k})$
$\curlywedge(0,\vec{\imath}^{\,k})$
$\curlywedge(1,\vec{\imath}^{\,k})$
$\curlywedge(n,\vec{\imath}^{\,k})$

Figure 1: Solution space. For each reconstructed index $\overrightarrow{\imath}^k$, the engine considers $(2n+1)$ possible values for $\overrightarrow{\imath}^{k+1}$. 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 $\curlywedge(3,\overrightarrow{\imath}^k) = [3,5,7,\mathbf{1}]$.

4 Experimental Evaluation

Our tool is written in Python and was used to extract codes from traces generated by the PolyBench/C 3.2 suite [Pou]. For illustrative purposes, Fig. 2a shows the source code of `cholesky` and Fig. 2b an excerpt of the memory trace generated by the access `A[i][k]` (see line 11 of Fig. 2a). Trace sizes and processing times are shown in Fig. 2c for the "standard" problem size, which generates up to 12.9 billion references in `3mm` (270 GB). Processing times largely depend on the number of reconstructed loops, as well as on the iteration pattern: the best efficiency is achieved when recognizing elements separated by the same stride. An exhaustive evaluation, including the crucial importance of our optimizations, as well as potential applications of the technique are detailed in [RAKT16].

References

[BHRS08] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of Programming Language Design and Implementation (PLDI)*, 2008.

[GGL12] T. Grosser, A. Größlinger, and C. Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.*, 22(04), 2012.

[Pou] L.-N. Pouchet. PolyBench: the polyhedral benchmark suite. `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`.

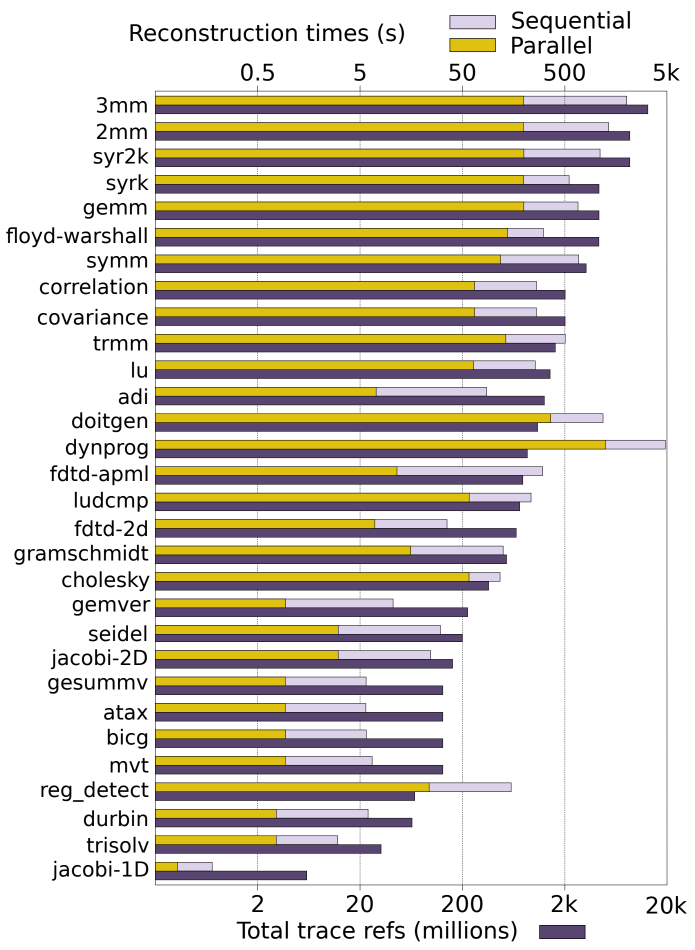
[RAKT16] G. Rodríguez, J. M. Andión, M. T. Kandemir, and J. Touriño. Trace-based affine reconstruction of codes. In *Proc. of Code Generation and Optimization (CGO)*, 2016.

```
 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 }
```

(a) 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
   ...
```

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



(c) Reconstruction times (upper axis) and trace sizes (lower axis). Axes are logarithmic. Since the subtraces are independent, they can be reconstructed in parallel achieving an average speedup of 5.6x. Each execution was performed on an Intel Xeon E5-2660 Sandy Bridge 2.20 Ghz node, with 64 GB of RAM.

Figure 2: Experimental evaluation with the PolyBench/C 3.2 suite [Pou].