Building of a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of non-Affine Programs Scalable

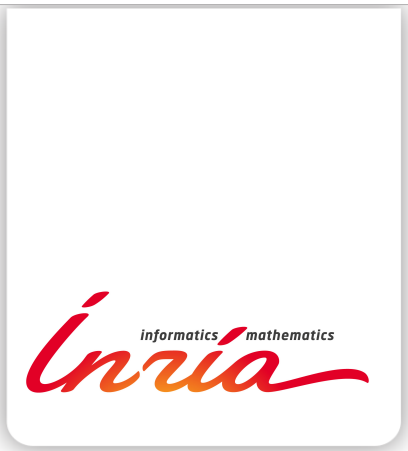
Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Louis-Noël Pouchet, Fabrice Rastello

Building of a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of non-Affine Programs Scalable

Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Louis-Noël Pouchet, Fabrice Rastello

RESEARCH REPORT
N° 9244
December 2018
Project-Team CORSE

ISSN 0249-6399 ISRN INRIA/RR--9244--FR+ENG



Building of a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of non-Affine Programs Scalable

Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Louis-Noël Pouchet, Fabrice Rastello

Project-Team CORSE

Research Report n° 9244 — December 2018 — 24 pages

Abstract: The polyhedral model has been successfully used in production compilers. Nevertheless, only a very restricted class of applications can benefit from it. Recent proposals investigated how runtime information could be used to apply polyhedral optimization on applications that do not statically fit the model. In this work, we go one step further in that direction. We propose a dynamic analysis that builds a compact polyhedral representation from a program execution. It is able to accurately detect affine dependencies and fixed-stride memory accesses in programs. The analysis scales to real-life applications, which often include some non-affine dependencies and accesses in otherwise affine code. This is enabled by a safe fine-grain polyhedral over-approximation mechanism applied to each analyzed expression. We evaluate our analysis on the entire Rodinia benchmark suite, enabling accurate feedback about potential for complex polyhedral transformations.

Key-words: Dynamic analysis, Polyhedral representation, Loop optimization, Polyhedral optimization, Trace compression,

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Création d'une Représentation Polyédrique depuis une Exécution

Résumé : Le modèle polyédrique est aujourd'hui utilisé à grande échelle via son intégration dans des compilateurs très largement utilisés. Néanmoins, seule une classe très restreinte de programmes peut en bénéficier. Des travaux récents ont montré comment des informations provenant d'une exécution du programme pouvaient être utilisées afin d'étendre la portée du modèle polyédrique. Ce travail s'inscrit dans ce contexte d'analyse dynamique de programmes pour appliquer le modèle polyédrique plus largement. Nous proposons une analyse dynamique capable de construire une représentation polyédrique d'un programme à partir d'une éxecution instrumentée. Cette analyse détecte de façon précise les dépendances affines ainsi que les accès mémoire avec incréments constants présents dans le programme. Notre analyse passe à l'échelle sur de vraies applications qui contiennent souvent quelques dépendances et accès mémoire non affines. Ce passage à l'échelle est possible grâce à un mécanisme de sur-approximation. Nous évaluons notre analyse sur la suite de benchmarks Rodinia en montrant quel est le retour fourni à l'utilisateur en ce qui concerne de potentielles transformations polyédriques.

Mots-clés : Analyse dynamique, Représentation polyédrique, Optimisation de boucle, Optimisation polyédrique, Compression de trace

1 Introduction

The most effective program transformations for improving performance or energy consumption are typically based on rescheduling of instructions so as to expose data locality and/or parallelism. The two main challenges for those kinds of optimizations are *what* transformations may be applied without breaking the program semantics and *where* in the program should the optimizations be applied so as to maximize the impact on the overall program performance. The polyhedral model [14], usually applied to static compilation, is a powerful tool for finding and applying such rescheduling optimizations. Compilers using the polyhedral model [32, 15] leverage precise information about data and control-flow dependencies to determine a sequence of loop transformations. These loop transformations aim to improve temporal- and spatial locality and uncover both coarse (i.e., thread) and fine-grain (i.e., SIMD) parallelism.

In practice, the static analysis used to recover the dependence information required for these transformations can only be applied to programs written in a very restrictive style with no function calls, only arrays as data structures, only very simple conditional statements and no indirections [11]. Dynamic analysis frameworks [12] address this limitation by reasoning on a particular execution of the program. The feedback provided by existing frameworks mainly informs about the absence of dependencies along some loop in the original program, highlighting opportunities for parallelism [20, 35, 34, 31] or SIMD vectorization [18].

In this work, we propose a dynamic analysis that does not merely prove the absence of dependencies, but recovers their structure in a form that fits the polyhedral model. We call this analysis the *folding-based analysis*. It can accurately detect polyhedral dependencies in programs. The analysis scales to real-life applications, which often include some non-affine dependencies in otherwise affine code. For that, we propose a safe fine-grain polyhedral over-approximation mechanism for these dependencies. That is, our analysis emits a compact program representation allowing a classic polyhedral compiler to find a wide range of possible transformations. Our analysis also allows detecting the presence of fixed-stride memory accesses. This information is useful for exposing potential for vectorization and loop transformations to improve spatial locality. The contribution of this paper is the folding-based analysis which:

- builds a compact polyhedral program representation from a program execution, enabling polyhedral compilers to be applied, that is, to provide feedback about the potential of complex polyhedral transformations [16];
- captures information useful for polyhedral optimizers such as properties of dependencies, data flow, memory accesses, and scalar evolution in a uniform manner; and
- scales to real-life applications by widening of non-affine expressions with a safe polyhedral over-approximation.

This paper is organized as follows. Section 2 illustrates the context of our work through a case study. It then continues with an in-depth description of the interface in Section 3, followed by the core algorithm used by our analysis in Section 4. Section 5 evaluates our approach by applying it to the entire Rodinia [9, 10] benchmark suite. Section 6 discusses related work. Finally, Section 7 concludes the paper and provides future perspectives.

2 Motivational Scenario

This section introduces the problem tackled by this work using a concrete example. For this, we use `backprop`, a benchmark from the Rodinia benchmark suite [9, 10]. `backprop` is a supervised

```
1 for (j = 1; j <= n2; j++) { // For each unit in second layer
2   float sum = 0.0; // Compute weighted sum of its inputs
3   for (k = 0; k <= n1; k++)
4      sum += conn[k][j] * l1[k];
5   l2[j] = squash(sum); }
```

Figure 1: A compute intensive kernel in `backprop`

learning method used to train artificial neural networks. We choose this benchmark because it is relatively short and simple, and still has behavior that cannot be handled accurately by existing analysis. For illustrative purposes, we focus on the compute kernel shown in Fig. 1. This kernel is also used as a running example throughout the rest of the paper.

As detailed below, while the source code appears affine, in fact taken in context of the full program, a compiler would assume overly pessimistic may-alias dependences due to how arrays are allocated. Dynamic analysis is required here to perform accurate polyhedral optimization on this kernel. To that end, we have developed a complete profiling tool-chain [16]. This tool-chain works on compiled binaries and provides suggestions for loop transformations. It is made of three parts. The *front-end*, which instruments the binary to get information from an execution of the program. The *folding-based analysis*, which consumes this information in a streaming fashion to build a compact polyhedral program representation along with information on the behavior of memory accesses. And finally, the *back-end*, which uses this representation to find interesting loop transformations. This paper focuses on the *folding-based analysis*, while the front-end and back-end have been presented in prior work [16].

2.1 Example problem: `backprop`

Back to Fig. 1, at first glance, all loop bounds and memory addresses seem to be affine expressions of loop invariant parameters, `n2` and `n1`, and enclosing loop iterators, `j` and `k`. This kernel should thus be a perfect target for polyhedral compilers such as LLVM-Polly [15] or GCC-Graphite [32]. Nevertheless, this is not the case because the `conn` object is not a two-dimensional array, but an array of pointers, each allocated by a separate call to `malloc` as illustrated in Fig. 2. Because of that, the compiler has to generate two load instructions for the access `conn[k][j]` as shown in Fig. 4. The first one loads `conn[k]` into a temporary variable `tmp` and the second one loads the value of `tmp[j]` into another temporary used in the multiplication. In other words, while one would expect a static compiler to see a single affine access to a two-dimensional array, there are actually two indirect accesses. Another problem is that it is impossible to know statically if the pointers `conn[k]`, `l1`, or `l2` alias, that is, whether they refer to the same object at runtime. Due to this, a static compiler has to conservatively assume there is a dependence between the write on Line 5 and the reads on Line 4. This dependence prevents any transformation of the outer loop. There is, furthermore, a function call on Line 5 which has to be inlined since it might hide other memory accesses.

Note that, `backprop` from Rodinia is not a real application but a simplified benchmark that does not interleave calls to `malloc` and `free`. Consequently `conn[0]`, `conn[1]`, ..., `conn[n1]` often happen to be laid out contiguous in memory. This is not realistic in practice, though. By introducing even only a short sequence of calls to `malloc` and `free` at the beginning of the benchmark the layout will be non-contiguous as shown in Fig. 2. The reason for this is that `malloc` gives no guarantees on the placement of allocations. In practice even runtime polyhedral

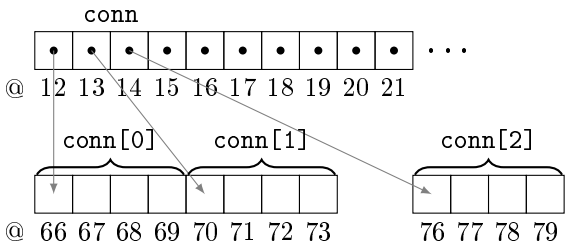


Figure 2: Memory layout for the `conn` array with `n2` = 3

optimizers [25, 29] will only sees random patterns when monitoring memory addresses for the access of `conn[k][j]` and perform no transformation.

2.2 Solution: folding-based analysis

Despite the presence of a non-affine memory access, the above computation kernel presents an interesting opportunity for optimization. Our dynamic analysis detects:

- the stride-1 access for `conn[k][j]` along the outer dimension j;
- the absence of dependence along dimension j;

From this information, our back-end suggests to perform a loop interchange, vectorization, and tiling. Applying those transformations lead to a speedup of $\times 5.3$.

The vectorization opportunity is revealed by looking at the scalar evolution [27, 33] of the addresses being accessed, that is, how they change as a function of the values of the iterators `k` and `j`. In the case of our example, the addresses used for loading `conn[k]` as shown in Fig. 2, can be described with the expression $0j + 1k + 12$, where 12 is the base address of `conn`. This is because `&conn[k]` does not depend on `j`, and `k` is incremented by one on each iteration. Note that due to the gap in the layout of `conn`, the addresses used to access `conn[k][j]` cannot be described by an affine expression. This is shown in Fig. 3. Our analysis is robust against this irregularity along dimension `k` and is able to produce the expression $1j + \top k + 66$, where 66 is the base address of the nested array `conn[0]`, and where $\top$ represents the fact that accesses are not affine along dimension `k`. However, the obtained expression does indicate that the memory address increases by 1 every iteration of dimension `j`. We refer to this as a *stride-1* access.

The folding algorithm not only discovers the structure of memory accesses, but also the structure of data dependencies in general. In our running example it detects the reduction in `sum` on line 4 of Fig. 1. It also detects that there is no dependence between the reads on line 4 and the write on line 5. It is worth mentioning that the structure of memory accesses and dependencies are detected separately. The folding algorithm can thus handle cases where accesses are non-affine and dependencies are affines. Here the irregularity of the former does not hinder the folding algorithm from finding the structure of the latter, and vice versa.

The stride-1 access along dimension `j` allows deducing that SIMD vectorization might be profitable. Since `j` is not the innermost loop it is necessary to perform a loop interchange before vectorizing. That this loop interchange is valid is clear from the absence of dependencies between the two loops. Note that the interchange will require an array expansion of the `sum` variable along with a new 1-dimensional loop iterating over `j` to fill the `12` array. Our analysis, like any dynamic approach reasoning on an execution, cannot guarantee that this holds in general, but it can still provide useful feedback.

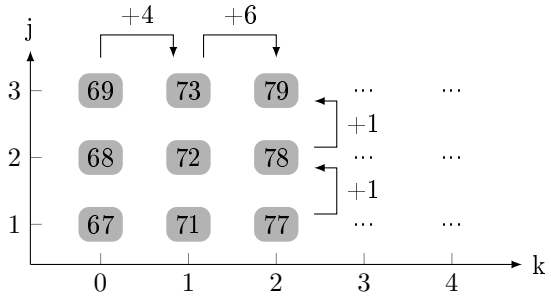


Figure 3: Addresses used to access `conn[k][j]`

3 Interface of the Folding-Based Analysis

Before describing the core algorithm of the folding-based analysis in Section 4, we introduce its inputs and outputs.

3.1 Inputs

The input for the folding-based analysis is provided by our front-end. In order to handle any kind of loops in a uniform way, our front-end inserts *canonical* iterators in every loop. These iterators start at zero and advance by one every iteration. The front-end is implemented as a plugin for the dynamic binary translator QEMU [3, 17]. Even though the front-end analyzes machine code it works at the level of the generic QEMU IR, making it CPU architecture agnostic.

The input of the folding algorithm is composed of streams of two types, one for instructions and one for data dependencies. In the following a *static instruction* is a machine instruction in the program binary. An *instruction instance* is one dynamic execution of a static instruction. A dependence is a pair consisting of an instruction instance that produced a value and another instance consuming it. We call those instances the *source* and the *destination* respectively. We note this as *source* $\rightarrow$ *destination*. The folding-based analysis itself does not depend on this and could work at any granularity, such as the source level.

Each input stream has a unique *identifier Id*. An instruction stream is identified by a static instruction, while a stream of data dependencies is identified by a pair of static instructions. The two types of streams have the same overall structure where each entry consists of two elements:

- an *iteration vector* (IV): a vector made up of the current values of all canonical loop iterators;
- a *label*: the definition of the label differs between the two types of streams and is described below.

For a given stream and a given identifier, all the IVs span a multi-dimensional space where each entry is a point. Thus, in the following we use the terms *entry* and *point* interchangeably. Also, note that IVs arrive in the input stream in lexicographical order.

Instructions An instruction stream for a static instruction *Id* contains all its instances. The label is a scalar value whose meaning depends on the type of the static instruction. If the instruction is an arithmetic instruction (Cpt) dealing with integers, the label is the integer value representing the result computed by the instruction. If the instruction is a memory access (Mem), the label is the address read or written by the instance.

```
1  for (j = 1; j <= n2)
2    sum = 0.0;
3    for (k = 0; k <= n1)
4      tmp1 = load(&conn + k)          I1 - Memory access
5      tmp2 = load(tmp1 + j)           I2 - Memory access
6      tmp3 = load(&l1 + k)            I3 - Memory access
7      sum = sum + tmp2 * tmp3         I4 - Computation
8      k = k + 1                       I5 - Computation
9    j = j + 1                         I6 - Computation
```

Figure 4: C-like binary version for the code of Fig. 1

Id=I1, Mem		Id=I2, Mem		Id=I4, Cpt		Id=I5, Cpt	
IV (cj,ck)	Label	*IV* (cj,ck)	Label	*IV* (cj,ck)	Label	*IV* (cj,ck)	Label
(0,0)	12	(0,0)	67	(0,0)	N/A	(0,0)	1
(0,1)	13	(0,1)	71	(0,1)	N/A	(0,1)	2
...	...	...	...	...	...	...	...
(0,41)	53	(0,41)	198	(0,41)	N/A	(0,41)	42
(1,0)	12	(1,0)	68	(1,0)	N/A	(1,0)	1
(1,1)	13	(1,1)	72	(1,1)	N/A	(1,1)	2
...	...	...	...	...	...	...	...

Table 1: Instruction input streams from example in Fig. 4

To illustrate the contents of the input stream of instruction instances we again use the example of `backprop` from Fig. 1. At the binary level, the considered loop-nest contains several instructions that are represented in an abstract C-like fashion in Fig. 4. An excerpt of four instruction streams for this example is shown in Table 1. The *IV* of each entry is the vector made up of the current values of all canonical loop iterators noted `cj` and `ck` in the table.

Dependencies A dependence stream for a pair of static instructions *source* and *destination* contains an entry for each pair of instances for these instructions that have a data dependence. The *IV* of an entry is the *IV* of *destination*, whereas the label is the *IV* of *source*. Table 2 shows the dependency input streams of the folding algorithm for the example in Fig. 4. In this example, all the dependencies except I4 → I4 are intra-iteration dependencies.

I1 → I2		I2 → I4		I4 → I4	
IV (cj,ck)	Label (cj',ck')	*IV* (cj,ck)	Label (cj',ck')	*IV* (cj,ck)	Label (cj',ck')
(0,0)	(0,0)	(0,0)	(0,0)		
(0,1)	(0,1)	(0,1)	(0,1)	(0,1)	(0,0)
...	...	...	...	...	...

Table 2: Dependency input stream from example in Fig. 4

Id	Polyhedron `(cj,ck)`	Label expression `f(cj,ck)`
I1	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$0cj + 1ck + 12$
I2	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$1cj + \top ck + 67$
I4	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	N/A
I5	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$0cj + 1ck + 1$

Table 3: Output of the folding algorithm for the instructions stream shown in Table 1 with `n2` = 16 and `n1` = 42

3.2 Outputs

The folding algorithm processes each stream independently. For each stream, the final result of folding is a piecewise linear expression mapping IVs to labels. We refer to this piecewise linear expressions as a *label expression*. The domain of a label expression contains all the IVs of all points seen in the input stream. We use the terms domain and *geometry* interchangeably in the following. Each piece of the domain is described by a set of affine inequalities, hence it defines a *polyhedron*. Section 2.2 already showed two examples of label expressions, e.g., $1j + \top k + 66$. This is a compact representation of the input stream since it can describe arbitrarily many points in one piece. It also directly exposes regularity in a form that polyhedral optimizers can exploit.

The coefficients of a label expression may be either an integer or $\top$, as illustrated for the non-affine memory access of `backprop` shown in Fig. 3. If a coefficient is $\top$, this indicates that the evolution cannot be expressed as an affine expression along the corresponding dimension.

Instruction For an instruction stream, depending on the type of its corresponding static instruction, the label expression either represents the integer values computed by the instruction or the addresses it accesses. Table 3 illustrates the outputs for the input streams in Table 1 where `n2` = 16 and `n1` = 42. All instruction instances of every input stream are now described by a single line each. We notice from this table that two of the four instructions have an affine expression where all the coefficients are known, that is, they are not $\top$. The affine expression of instruction I4 is marked as N/A because it is computing floating point values. Instruction I2 has an affine expression with the coefficient for dimension k being $\top$, as already discussed. Nevertheless, the algorithm still outputs the single polyhedron describing the domain for this instruction. It is also worth mentioning that, unlike in this example, the label expression of each instruction can be made up of several pieces. The domain would then be represented as a union of polyhedra.

Dependencies The label expression of a dependency is a piecewise linear expression with multiple outputs. The label expression maps IVs of the consumer instances of the dependence to IVs of the producer instances. That is, given an instruction instance the label expression can be used to determine from which other instruction instances it consumed data. Table 4 illustrates the result of the folding-based analysis for the three dependency input streams in Table 2. All the dependencies of a given input stream are now described by a single line. Each one of these lines states when the dependency between two instruction instances occurs. For example, the last line tells us that the instance (cj, ck) of I4 depends on the instance $(cj, ck - 1)$ of itself. As for the output regarding instruction streams, it is worth noting that in this example the domain of all the dependencies is described by a single polyhedron. Nevertheless, in more complex cases these domains can be represented by a union of polyhedra.

Id	Polyhedron `(cj,ck)`	Label expression `f(cj,ck)`
I1 → I2	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$cj' = cj + 0ck,\ ck' = 0cj + ck$
I2 → I4	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$cj' = cj + 0ck,\ ck' = 0cj + ck$
I4 → I4	$0 \leq cj \leq 15, 1 \leq ck \leq 42$	$cj' = cj + 0ck,\ ck' = 0cj + ck - 1$

Table 4: Output of the folding algorithm for the dependencies stream shown in Table 2

3.3 Using the output

The output of the folding algorithm is intended to be consumed by the back-end of our tool chain leveraging a classic polyhedral optimizer. Such an optimizer requires as input the list of instructions along with their domains and their dependencies. The back-end then searches which re-scheduling transformations can be applied to the instructions under the constraints imposed by the data dependencies.

Before providing dependencies to the back-end, the output stream of dependencies is pruned by removing all the dependencies involving a computation instruction identified as an *induction variable*. An induction variable is a computation instruction with a label expression where all coefficients of all pieces are integers, that is, not $\top$. The initial loop iterators are an example of induction variable, e.g., I5 and I6. Removing those instructions serves two purposes. First, induction variables always depend on their value from the previous iteration of the loop they are in. Consequently their dependencies constrain the execution to be completely sequential. Removing these instructions gives the back-end more freedom and may uncover parallelism or potential for other polyhedral transformations. The second reason for removing induction variables is simply that it reduces the number of instructions the polyhedral back-end has to deal with.

Then, still before providing the dependencies to the optimizer, we must process dependencies having $\top$ coefficients in their label expression. Observe that the fact that some dependencies are not accurately captured by our folding algorithm is not a limitation of the approach, but a choice imposed by polyhedral back-ends which complexity are combinatorial with the size of the polyhedral representation. To that end, we over-approximate those dependencies by imposing a lexicographical ordering over their IVs for the iterators having at least one *top* coefficient. With this order, it is guaranteed that all instances of the producer come before any instances of the consumer that might possibly consume them. For instance, let us assume in our running example that the dependency I4 → I4 is not $cj' = cj + 0ck,\ ck' = 0cj + ck - 1$ but $cj' = cj + 0ck,\ ck' = 0cj + \top ck$: The over-approximated dependency given to the back-end would be $cj' = cj \wedge ck' \leq ck$. Note that, if for the sake of clarity the widening described by our folding algorithm is rather extreme, i.e., the value in our lattice is either an integer or $\top$. Intermediate values such as intervals could be used instead. E.g., see [7] for a description of different dependency levels.

Finally, the access functions for memory instructions are also given to the polyhedral optimizer so that it can identify opportunities for exposing vectorization and spatial locality. For this it needs information about stride which is given by a non-$\top$ coefficient in the label expression of an instruction accessing memory.

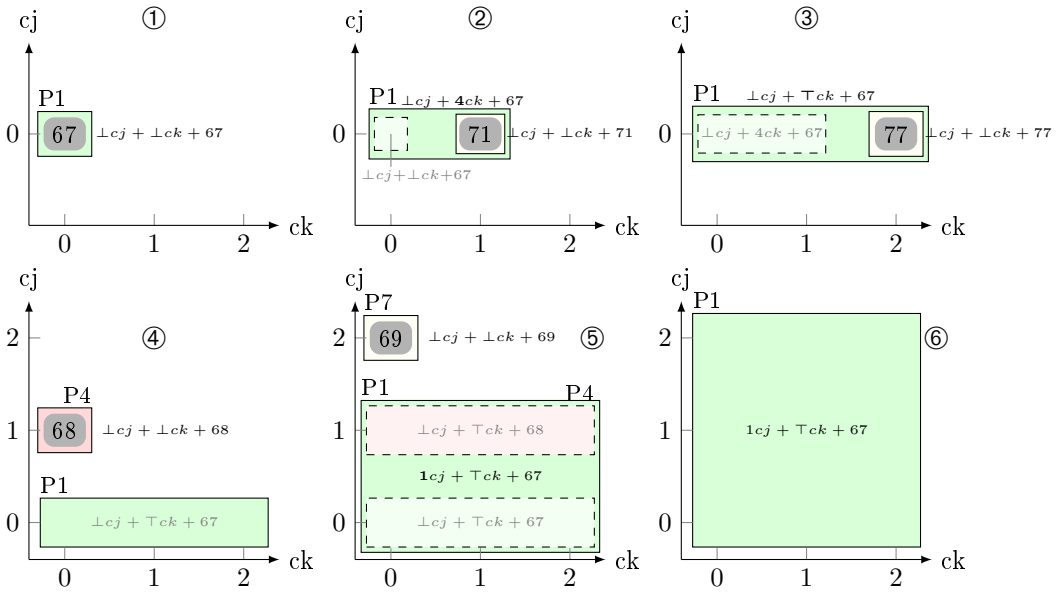


Figure 5: Folding process for the input stream in Fig. 3 considering only three points in both dimensions.

4 The Folding Algorithm

This section gives an overview of the folding algorithm and then presents the details of the algorithm.

4.1 Overview

As stated in the previous section, the folding algorithm processes the stream for each identifier separately. For both instruction and dependency streams, the algorithm receives points in a geometrical space as specified by the IVs. In both cases, the main idea of the algorithm is to construct polyhedra from those points. For each polyhedron the algorithm constructs an affine expression describing the label of the points contained in the polyhedron. When receiving the first point, the algorithm creates a 0-dimensional polyhedron containing only that point. It then tries to grow this polyhedron with the next points, adding dimensions as necessary.

To give an intuition about how the folding algorithm works, let us consider the stream of I2 in Table 1.

4.1.1 Geometric folding

The folding process for I2 is illustrated in Fig. 5. For now we will ignore the construction of the affine expression. As shown, the process leads to the creation of many intermediary polyhedra which are merged as the algorithm executes. The polyhedron P1, a 3×3 square, is the final result of the algorithm. As shown in Fig. 5 the main steps of the algorithm are as follows:

① create the 0-dimensional polyhedron P1 when the first point $(cj = 0, ck = 0)$ is received;

② when $(cj = 0, ck = 1)$ is received, P1 absorbs it to become a 1-dimensional polyhedron, that is, a line segment;

③ when $(cj = 0, ck = 2)$ is received, P1 absorbs it;

④ notice that the loop over ck is completed when point $(cj = 1, ck = 0)$ is received because the iterator of the surrounding loop cj increased. Then create the new 0-dimensional polyhedron P4;

- P4 absorbs $(cj = 1, ck = 1)$ to become a 1-dimensional polyhedron and then absorbs $(cj = 1, ck = 2)$ (not shown in Fig. 5);

⑤ notice that the loop over ck is completed when point $(cj = 2, ck = 0)$ is received. P1 absorbs P4 along dimension cj. Then create the new 0-dimensional polyhedron P7;

- P7 absorbs $(cj = 2, ck = 1)$ to become a 1-dimensional polyhedron and then absorbs $(cj = 2, ck = 2)$ (not shown in Fig. 5);

⑥ P1 absorbs P7 and becomes the final 3×3 square.

The geometric folding works exactly the same for dependencies as illustrated above for instructions. The only difference is the semantic of the reconstructed union of polyhedra. In the case of an instruction, this union defines when the instruction is executed. For a dependency it tells when the dependency occurs from the point of view of the destination.

4.1.2 Label folding

In the previous section we ignored the folding of the labels associated with each point in the input stream. Nevertheless, this label folding takes place at the same time as geometric folding. It is also performed in a streaming fashion. In the context of label folding, the symbol $\perp$ denotes a coefficient that has not yet been determined because the loop has not yet iterated along the dimension associated with that coefficient. As shown in Fig. 5 the label folding proceeds as follows:

① create $f1(cj, ck) = \perp cj + \perp ck + 67$ when point $(cj = 0, ck = 0)$ with label 67 is received;

② update $f1$ to $\perp cj + \mathbf{4}ck + 67$ when P1 absorbs $(cj = 0, ck = 1)$ with label 71 is received because ck advanced by 1 and $71 - 67 = 4$;

③ check if $f1(cj, ck) = \perp cj + 4ck + 67$ is valid when P1 absorbs $(cj = 0, ck = 2)$ with label 77. It is not the case, so update $f1$ to $\perp cj + \top ck + 67$;

- repeat the steps above for P4 and get $f4(cj, ck) = \perp cj + \top ck + 68$ (not shown in Fig. 5);

⑤ update $f1$ to $f1(cj, ck) = \mathbf{1}cj + \top ck + 67$ when P1 absorbs P4 because cj advanced by 1 and $68 - 67 = 1$;

⑥ check whether $f1(cj, ck) = 1cj + \top ck + 67$ is compatible with $f7(cj, ck) = \perp cj + \top ck + 69$, when P7 absorbs P1 to get the final 3×3 square. It is the case.

The algorithm that folds the labels of a dependency is the same as the one described above for the label of an instruction. It is just applied *individually* for each scalar value in the label vector, that is, each component of the IV of the source of the dependency.

4.2 The algorithm

This section introduces the structure of the main algorithm itself and then explains its subcomponents.

4.2.1 Main folding function

The main function is shown in Algorithm 1. As explained in Section 3, this main function, is applied to each input stream separately. To handle real-life applications, where input streams are huge, the algorithm works in a streaming fashion (Line 10). It is not necessary to have the whole input available at once. The output is also emitted as a stream. The main principle of the algorithm, as depicted in the example in Fig. 5, consists of maintaining a worklist of *intermediate* polyhedra per dimension. The intermediate polyhedra then grow by absorbing other polyhedra. Note that a d-dimensional polyhedron can only absorb $(d-1)$-dimensional polyhedra.

Representation for polyhedra The folding algorithm only produces bounded convex polyhedra (polytope). Internally, these polyhedra are represented by their extreme points (vertices). Also, all the polyhedra have edges whose "slopes" are always in $\{-1, 0, 1\}$. We call those, *elementary polyhedra*, and define them formally using the following recursive definition:

- an elementary 0-dimensional polyhedron is a polyhedron made of a single point;
- an elementary d-dimensional polyhedron is a convex polyhedron with 2^d extreme points such that: 1. All its extreme points must have identical coordinates in dimensions higher than d; 2. Its lower and upper faces must themselves be $(d-1)$-elementary polyhedra; 3. The edges connecting the lower and upper faces can be expressed as $k\vec{S}$. Where $k \in \mathbb{N}^*$ and $\vec{S}$, the *slope vector* of the edge, is a vector where all components are either -1, 0, or $+1$.

A polyhedron is *degenerate* on a given dimension if all its extreme points have the same coordinate for that dimension, that is, it has zero width in that dimension. The elementary polyhedra produced by the folding algorithm may be degenerate on one or more dimensions.

Producing only elementary polyhedra allows the absorption process described below to work in no more than $O(3^{d-1})$ time. The choice of producing only such polyhedra is also motivated by the nature of the input streams that we want to process. The front-end we use to feed the folding algorithm always produces IVs starting at zero and only ever advancing by one. Hence, elementary polyhedra are able to represent the behaviour of most regular loops.

Data structures Folding works on spaces with a fixed number of dimensions d, that is, the dimensionality of the corresponding IVs. The state of the folding algorithm is contained in two dictionaries. The first one, `absorbers` (Line 2), contains a list of intermediate polyhedra for each dimension. `absorbers[d]` only contains d-dimensional, potentially degenerate, polyhedra. The polyhedra in `absorbers[d]` are those that can still grow along dimension d by absorbing $(d-1)$-dimensional polyhedra. Those $(d-1)$-dimensional polyhedra are stored in `eps_2_to_be_absorbed[d]` (Line 6). The keys of the dictionary `eps_2_to_be_absorbed[d]` are the lexicographically first, extreme points of the polyhedra to be absorbed. This point, which we name the *anchor*, is used to uniquely identify the absorbed polyhedron. The `abso.upper_left` (Line 27) is the extreme point from which the absorption is done. This point is the lexicographically first point of the upper face of `abso`.

Analysis steps When a point is received, the algorithm first processes the innermost dimension (numbered 1). Then, for each loop (but for the outermost one) that completes in the instrumented code, the algorithm processes its enclosing dimension. In other words, if the innermost loop finishes, the algorithm processes dimension $d = 2$; if its enclosing loop finishes, it processes dimension $d = 3$; etc. In Line 17, `process_dims` represents that set of dimensions to be processed.

Before processing the different dimensions, the current point is added into `absorbers[0]` (Line 14). This state is only transient, because as soon as the innermost dimension is processed, the point will be promoted into `eps_2_to_be_absorbed[1]` (Line 21). Then, for each dimension d of `process_dims` (processed from inner to outer), three steps are performed.

The first step (Lines 20 to 21) promotes all polyhedra in `absorbers[d-1]` into `eps_2_to_be_absorbed[d]`. Because dimensions are processed in increasing order, that is, from innermost to outermost, when processing dimension d we are sure that `absorbers[d-1]` have already absorbed all the $(d-2)$-dimensional polyhedra it could. This promotion to d-dimensional degenerate polyhedra allows them to be absorbed in the next step by the d-dimensional polyhedra already in `absorbers[d]`.

In the second step (Lines 24 to 39), polyhedra from `absorbers[d]` try to absorb polyhedra in `eps_2_to_be_absorbed[d]`. For absorption to be possible, the polyhedra should be geometrically compatible (Line 30) and their label expressions should match (Line 31) as described in Section 4.2.1 and Section 4.2.2. If a polyhedron in `absorbers[d]` does not absorb any other polyhedron, then it will never grow again along dimension d. As a consequence, it is promoted into `eps_2_to_be_absorbed[d+1]` (Line 39). This promotion also transforms the d-dimensional polyhedron into a $(d+1)$-dimensional degenerate polyhedron.

The third and last step (Lines 43 to 44) promotes all the d-dimensional polyhedra in `eps_2_to_be_absorbed[d]` that have not been absorbed. Since those polyhedra will never be absorbed again in dimension d, they are moved to the `absorbers[d]` list so that they will have a chance to themselves absorb other polyhedra next time dimension d is processed.

During the execution of the algorithm, a polyhedron is *retired* when it is promoted to the dimension above the maximum dimension of the space, e.g., 3 for an instruction in a 2D loop nest. When the stream is finished, all remaining non-retired polyhedra are also retired. Retired polyhedra are written to the output stream and do not consume memory anymore. This is safe since we know that they will never grow anymore.

4.2.2 Absorption

As stated in Section 4.1, the second step of the folding algorithm grows polyhedra by letting them absorb each other. A d-dimensional polyhedron searches for candidates to absorb by checking if the first point of its upper face, called the *corner* (Line 27), touches the anchor of any other $(d-1)$-dimensional polyhedron. This search is done by adding the *search vectors* `v` to the coordinates of the corner and performing a lookup in `eps_2_to_be_absorbed[d]` to see if there is a polyhedron at this position (Line 28). Once a candidate has been found, the algorithm must check that all other extreme points also match (Line 30). Which search vectors are used for this lookup and how geometric matching is checked depends on whether the absorber is degenerate in d or not. If the absorber is degenerate we call this a *polyhedra merge*. An example of this is when P1 absorbs P4 in Fig. 5. The second case, a *polyhedra extension*, occurs when the absorber is not degenerate, as seen for example when P1 absorbs P7.

Polyhedra merge In this case, the d-dimensional absorber polyhedron is degenerate on dimension d. Hence, it has no slopes yet on that dimension. As a consequence the set of search vectors used to find candidates are all possible slope vectors where the value of the i^th component is *a)* 0 if $i > d$; *b)* 1 if $i = d$; *c)* in $\{0, -1, +1\}$ if $i < d$. A polyhedra merge is legal if every extreme point P of the upper face of the absorber can be connected to the corresponding extreme point of the polyhedron to be absorbed using any slope vector strictly greater than P. Note that there are always exactly 3^{d-1} such slope vectors and that they can be pre-computed. After absorption the resulting polyhedron will no longer be degenerate in d. Its lower face will be the lower face

of the original absorbing polyhedron and the upper face of the absorbing polyhedron will be the extreme points of the $(d-1)$-dimensional absorbed polyhedron. The slopes of edges between the faces of the new polyhedron will be exactly those search vectors used to connect the extreme points.

Polyhedra extension In this case the absorber is a non-degenerate d-dimensional polyhedron. Hence, the absorber already has slopes for all its edges. When looking for candidates to absorb, there is only one search vector, the slope vector connecting the oldest extreme point of the lower face to that of the upper face. To check if it is legal for the absorber to absorb the candidate it suffices to verify whether the extreme points of the two polyhedra can be connected using the slopes of the absorber.

After an absorber polyhedron has found a candidate with extreme points that match, the algorithm checks whether the labels also match. This is done by the `has_compat_label` function described in the next section. If both the geometry and the label are compatible, then the absorption is performed by a call to `absorbs`.

4.2.3 Compatibility and update of label expressions

The data structure used for label expressions is shown in Fig. 6. `num_dimensions` is the number of loops enclosing the static instruction or the destination instruction associated with the input stream.

```
Label_Function:
  int num_dimensions
  int[num_dimensions + 1] init_point
  int[num_dimensions + 1] coeffs
  coeff_t[num_dimensions + 1] coeff_types
```

Figure 6: The data structure used to represent label expressions

Creation Label expressions are created when a new polyhedron is created from a single point (Line 14). At this time, all the coefficients of the expression are still unknown. Their types in the `coeff_types` array are set to $\bot$. The coordinates of the point used to create the new polyhedron are saved in the `initial_point` array. The first cell of this array is never used but still kept to make accesses more readable, that is, `initial_point[d]` contains the d^{th} coordinate. These coordinates are used when coefficients are updated. Note that once a coefficient has been updated from an unknown to a known value, it is never updated again except to be set to $\top$. At creation time, `coeff[0]` is given the value associated with the initial point. As long as there are some $\bot$ coefficients, `coeff[0]` contains the remaining amount contributed by unknown coefficients. We refer to `coeff[0]` as the *remaining value* in the following. This remaining value is updated whenever a coefficient is updated. When all coefficients are known, the remaining value represents the constant coefficient of the affine expression.

The two polyhedra involved in a compatibility check along dimension d may be degenerate on one or more dimensions, including the d^{th} one. As a consequence, the check may be faced with affine expressions where some coefficients are $\bot$. In the following, we note the label expression of the absorbing polyhedron as `f_abs`, and that of the polyhedron to be absorbed as `f_to_be_abs`.

We notice that the polyhedron to be absorbed is always degenerate on dimension d, as stated in Section 4.2.1. Hence, `f_to_be_abs.coeff_types[d]` $= \perp$.

All dimensions below are known For illustrative purposes we first cover the simplified case where all dimensions below d are known for the two label expressions. Here the compatibility check, is straightforward. First the function `has_compat_label` verifies that all coefficients for dimensions from 1 to $d-1$ are the same. If this is not the case the two label expressions are incompatible, and it returns `false`.

Otherwise, the check may be faced with two cases corresponding to the two different absorption cases described in Section 4.2.2. In the polyhedra merge case, where the absorber polyhedron is degenerate on dimension d, that is, `f_abs.coeff_types[d]` $= \perp$, the check always succeeds and the `has_compat_label` function returns `true`. Indeed, by setting the proper coefficient for dimension d and by updating the remaining value, it is always possible to make the two expressions compatible as shown by the `update_label_dims_known` function in Algorithm 2. The new coefficient is equal to the difference of remaining values (Line 6). Note that, in general we would also have to divide the new coefficient by the progress made along dimension d. However, because absorption guarantees that the two polyhedra whose label expressions are being merged touch each other the progress is always equal to 1. Finally, the remaining value is decreased by the effective contribution of the new coefficient taking into account the d^{th} coordinate of the initial point (Line 10).

In the polyhedra extension case, the absorber polyhedron is not degenerate on dimension d. Its affine expression already has a value computed for the coefficient on dimension d Then `f_abs.coeff_types[d]` $\neq \perp$ and nothing needs to be updated. The compatibility check must only ensure that this coefficient is compatible with `f_to_be_abs`. Algorithm 2 shows the `has_compat_label_dims_known` function implementing this check. First, it computes the contribution of the known coefficient of `f_abs` into `f_to_be_abs` using the initial point of `f_to_be_abs` (Line 14). Then, the check subtracts this contribution from the remaining value of `f_to_be_abs` to compute its new remaining value. For the check to return `true`, this new remaining value must be equal to the remaining value of `f_abs` (Line 16).

General case In the general case the two polyhedra may be degenerate for some dimensions below d. This happens if a dimension only iterates once. The compatibility check described above must take this into account.

Function `has_compat_label_general` in Algorithm 3 shows the general compatibility check. If `f_abs` does not have a coefficient set for dimension d, the check always succeeds as in the particular case previously described. The check works by comparing the coefficients of both expressions for all the dimensions from 1 to d. If both coefficients for a dimension are known they must be the same or the check fails (Line 9). If one is known and not the other (Line 12 and Line 14), then the function increments the total contribution coming from the other expression for the expression having the unknown coefficient. At the end of the loop, the check ensures that the coefficient for dimension d in `f_abs` is compatible with `f_to_be_abs`. This check relies on the total contribution variables incremented during the loop to ensure that the two expressions still produce the same value after merging.

In case they are compatible, the new coefficients, that is, the one on dimension d and potentially others, and the new remaining value for the expression of the absorber are computed by the same principles as the ones performed by the `has_compat_label_dims_known`.

Label widening As shown by the `backprop` example, the folding algorithm must be capable of identifying labels that are affine on some dimensions and not on others. To that end, the

algorithm has a mechanism called *label widening* enabling it to skip the matching of labels on a per dimension basis. If the compatibility check between two coefficients fails, then instead of returning `false` (Line 11 in Algorithm 3), the coefficient is set to $\top$ and `true` is returned instead. The absorption can still happen, even if the labels of the two polyhedra are not fully compatible. The resulting polyhedron is no longer a fully accurate representation of the input stream. Nevertheless, this mechanism allows the folding algorithm to handle real life applications without a perfect affine behavior. The name label widening stems from the fact that in the case of dependencies it widens the label expressions from strict equalities to inequalities, as shown in Section 3.3.

The integration of this feature into Algorithm 2 and Algorithm 3 is straightforward. A $\top$ coefficient is compatible with any other coefficient, and when performing absorption, any such coefficient in one of the two label expressions leads to a $\top$ coefficient in the updated expression.

The label widening mechanism is crucial for the label expressions of instructions because $\top$ is a clear indicator that a memory accesses is not affine along a dimension. For dependencies it simply reduces the size of the output given to the back-end by reducing the number of produced pieces.

4.2.4 Geometric give up

Even with the label widening mechanism described above, some applications may lead to the creation of a huge number of polyhedra. This happens when the geometry of instructions and dependencies are not affine. In the worst case, the folding algorithm creates one polyhedron for each dynamic instruction and for each dynamic dependency.

To mitigate this issue, the folding algorithm has another global option called *geometric give-up*. This options allows defining an upper limit on the number of intermediate polyhedra. Remember that an intermediate polyhedron is a polyhedron in one of the worklists that can still grow by absorbing other polyhedra. Before creating a new polyhedron (Line 14), the algorithm checks if the number of intermediate polyhedra exceeds the threshold. If so, then the associated input stream is marked as *give up*. Once a stream has been marked as give up, nearly all information heretofore collected for it is discarded. First, all intermediate polyhedra and also the polyhedra that have already been retired are discarded. Furthermore, all coefficients for all outputs of the label expression are set to $\top$, that is, a geometric give up implies giving up on all dimensions of the label expression. The only information that is retained for the stream is the maximum coordinates seen in the IVs of any point. From then on every time a new point is received for the given up stream, the folding algorithm previously described is skipped. Instead, only the maximum coordinates seen are update as necessary for every points.

The final geometry emitted for a give up stream is simply an hyperrectangle that starts at the origin and extends to the maximum coordinates seen in the IVs of any point of the input. In other words, the geometry of the input stream is over-approximated by a large polyhedron.

5 Experimental Results

This section applies our analysis to a full benchmark suite to demonstrate the scalability of the folding algorithm and show that it extracts rich information for optimization.

Experimental setup We use the latest revision, 3.1, of the Rodinia benchmark suite [9, 10]. All measurements and experiments where performed on a Xeon Ivy Bridge CPU with two 6 core CPUs, each running at 2.1GHz. As the front-end producing the IVs and labels does not support multithreaded applications yet, each benchmark is run with a single thread. All

benchmarks where compiled using GCC 8.1.1. Since QEMU, which the front-end is based on, currently cannot handle newer AVX instructions we used the compiler flags `-g -O2 -msse3`. For the speedup measurements of `backprop` mentioned in Section 2.2 we used the Intel icc 18.0.3 compiler and the flags `-Ofast -march=native -mtune=native`.

Note that the instructions in our experiments are real X86 machine instructions. Many X86 instructions both read or write memory and perform computations at the same time. As a consequence the instructions streams that form the input of the folding algorithm are actually more complicated than the ones presented in Section 3.1 and Table 1 in a simplified way for clarity purposes. In reality the label of an instruction can have multiple values to account both for the addresses accessed and the values produced. The label expressions for instructions thus potentially have multiple outputs as well, just like those for dependencies.

Table 5 gives statistics on the size and precision of the output of four versions of the folding algorithm. $\boldsymbol{F}$ is the basic algorithm as described in Section 4, with label widening for instructions and without for dependencies. $\boldsymbol{F_W}$ is the algorithm with label widening for both instructions and dependencies. $\boldsymbol{F_{GG}}$ is the same as $\mathbf{F}$ but with geometric give up. $\boldsymbol{F_{GG,W}}$ is the same as $\mathbf{F_W}$ but with geometric give up. The treshold for the geometric give up was set to allow $4d+1$ intermediate polyhedra in each d dimensional space. That is, enough for the affine expression constructed to be made up of up to four d dimensional pieces.

For each algorithm we report the following statistics. **#P** is the number of polyhedra in the output stream. For dependencies, **%A** is the number of dependence instances that where in an affine piece of the label expression. A piece of the label expression is considered affine if it has no $\top$ coefficient. This column is omitted for algorithm $\mathbf{F}$ since by construction it always contains 100%. Similarly for instructions, **%A** is the number of instruction instances that where in an affine piece of the label expression. A piece of the label expression of a static instruction is considered affine if it either: (a) does not perform a memory access, or (b) has no $\top$ coefficient in its memory access function. $\mathbf{\#MP}_{\boldsymbol{l}}$ is the maximum number of intermediate polyhedra live at any moment of the execution, indicating the memory usage of the algorithm.

The remaining columns in the table are as follows. **Input Size** shows the total number of entries in all dependency and instruction input streams. **Optim** shows a very brief outline of the optimization feedback given by our polyhedral back-end using the output of $\boldsymbol{F_{GG,W}}$ [16]. In this column $\boldsymbol{T}$ $n\boldsymbol{D}$ indicates that the back-end has found that n dimensional tiling was possible. $\boldsymbol{P}$ indicates that the back-end has detected parallelism that can be exploited using threads. $\boldsymbol{V}$ indicates that the back-end has detected potential for vectorization. Note that the entire feedback of the tool is immensely richer and more elaborated [16], this column gives only a simplified summary.

Finally, note the numbers reported in Table 5 corresponds to applying the folding-based analysis on the hot region of each benchmark, we have filtered out the phases where the benchmarks read their input or write their output. This hot region often involves numerous function calls [16].

Discussion of the results Since the polyhedral optimization performed in the back-end is an exponential problem it is crucial that the output of the folding-based analysis is of tractable size. Table 5 clearly shows that $\mathbf{F_{GG}}$ and $\mathbf{F_{GG,W}}$ produce drastically smaller outputs than the other two versions. As indicated by the **%A** column, $\mathbf{F_{GG,W}}$ is roughly as precise as $\mathbf{F_{GG}}$, but produces an even smaller output. In fact only the output of $\mathbf{F_{GG,W}}$ is small enough for the back-end to handle.

Since Rodinia is a benchmark suite designed to exploit multi-core parallelism each benchmark contains at least one parallel loop. As seen in column **Optim** the folding-based analysis clearly detects this parallelism across the entire suite, even in the presence of may-alias dependencies in the source code. We also find that there is tiling potential across Rodinia.

Benchmark	Dependencies												Instructions							Optim
	Input Size	F #P	F #MP_l	F_W #P	F_W %A	F_W #MP_l	F_{GG} #P	F_{GG} %A	F_{GG} #MP_l	$F_{GG,W}$ #P	$F_{GG,W}$ %A	$F_{GG,W}$ #MP_l	Input Size	F_W #P	F_W %A	F_W #MP_l	$F_{GG,W}$ #P	$F_{GG,W}$ %A	$F_{GG,W}$ #MP_l	
backprop	19M	160	385	160	100%	385	160	100%	385	160	100%	385	15M	140	99%	304	140	99%	304	T 2D, P, V
bfs	5M	903K	965K	874K	93%	951K	74	31%	772	70	31%	772	4M	520K	82%	472K	38	51%	367	T 2D, P
b+tree	95M	91K	390K	86K	99%	336K	113	99%	3K	113	99%	3K	61M	50K	90%	153K	160	89%	1K	T 3D, P, V
cfd	782M	530	1K	525	98%	1K	530	100%	1K	525	98%	1K	498M	332	100%	961	332	100%	961	T 3D, P, V
heartwall	33G	3K	8K	2K	90%	6K	1K	10%	5K	1K	10%	5K	18G	1K	69%	3K	1K	9%	3K	T 5D, P
hotspot	19M	11K	22K	10K	95%	21K	785	0%	6K	785	0%	6K	11M	6K	71%	13K	520	0%	3K	T 2D, P
hotspot3D	235M	168	1K	162	91%	1K	168	100%	1K	162	91%	1K	183M	84	85%	782	84	85%	782	T 3D, P
kmeans	1G	135	477	131	99%	472	135	100%	477	131	99%	472	911M	82	95%	281	82	95%	281	T 4D, P, V
lavaMD	1G	7K	2K	7K	94%	2K	7K	100%	2K	7K	94%	2K	923M	4K	71%	1K	4K	71%	1K	T 3D, P
leukocyte	5G	516K	161K	514K	99%	113K	162	99%	66K	162	99%	65K	2G	355K	84%	72K	128	84%	40K	T 3D, P, V
lud	89M	2K	1K	2K	98%	1K	2K	98%	1K	2K	98%	1K	51M	1K	97%	864	1K	97%	864	T 3D, P
myocyte	4M	5K	9K	5K	100%	9K	5K	100%	9K	5K	100%	9K	3M	3K	99%	4K	3K	99%	4K	T 1D, P, V
nn	782K	124	242	124	100%	211	124	100%	241	124	100%	211	855K	160	100%	189	160	100%	189	T 1D, P
nw	217M	301	1K	296	99%	1K	301	100%	1K	296	99%	1K	111M	155	100%	555	155	100%	555	T 2D, P, V
particlefilter	3G	5K	92K	3K	99%	2K	550	8%	2K	541	8%	2K	2G	2K	99%	1K	474	11%	1K	T 2D, P, V
pathfinder	74M	35	139	35	100%	135	35	100%	139	35	100%	135	42M	24	61%	116	24	61%	116	T 2D, P
srad_v1	3G	250	851	242	94%	824	250	100%	851	242	94%	824	2G	179	93%	531	179	93%	531	T 2D, P
srad_v2	1G	276	811	268	97%	791	276	100%	811	268	97%	791	721M	204	93%	493	204	93%	493	T 2D, P
streamcluster	2G	1M	1M	1M	85%	1M	8K	85%	13K	6K	85%	12K	1G	611K	71%	618K	3K	71%	6K	-

Table 5: Evaluation of the folding algorithm

Note that `streamcluster`, the least affine of all benchmarks, exhausted memory in the polyhedral back-end and therefore no result is displayed. Benchmark `mummergpu` is not included in the results since it contains CUDA code and the front-end can only instrument code run on the CPU.

6 Related Work

Integer linear algebra is a natural formalism for representing the computation space of a loop nest. The polyhedral framework [14] leverages, among others, operators on polyhedrons, enumeration for code generation [2], and parametric integer linear programming [13] for dependence analysis [11]. Historically, it has been designed to work on restricted programming languages, and was used as a framework to perform source-to-source transformations. More recently, efforts have been made to integrate the technology in mainstream compilers with GCC-Graphite [32] LLVM-Polly [15]. The set of loop transformations that the polyhedral model can perform is wide and covers most of the important ones for exposing locality and parallelism to improve performance [6].

Dynamic data flow/dependence analysis is a technique typically used to provide feedback to the programmer, e.g., about the existence or absence of dependences along loops. The detection of parallelism along canonical directions, such as vectorization, has been particularly investigated [22, 8, 35, 20, 1, 21, 12, 34, 31], as it requires only relatively localized information. Another use case is the evaluation of effective reuse [24, 23, 5, 4] with the objective of pinpointing data-locality problems. Like us, with the objective of gathering a more global dependence information, Redux [26] builds a complete extended dynamic dependence graph from binary level programs. The paper concludes with a negative result. Because of its inability to compress the produced graph it is only able to handle very small non-realistic programs.

Among the existing trace compression algorithms, two specialize in extracting a polyhedral representation from input streams [19, 28]. However, although they excel in rebuilding a polyhedral representation for a purely affine trace, they suffer inherent limitations for (even partially) non-affine traces. They share the idea of using pattern matching with affine expressions with our folding algorithm but do not exploit the geometric information provided by the IVs. Non-geometry-based approaches require a finite window of points under consideration. Unfortunately, this forces a trade-off between speed for and quality of the output when choosing the size of this window. For perfectly regular programs a small window can be used, making the algorithms

very efficient. In that simple case using a geometric approach does not make much difference. With d the dimension of the iteration space and n the number of points, the complexity of both non-geometric approaches is $O(2^d n)$. However, in the context of profiling large non-fully affine programs, none of these two existing approaches can be used. The complexity of the nested loop recognition algorithm of Ketterlin et al. [19] increases quadratically with a parameter k that bounds the size of the window. If k is smaller than the amount of irregularity along the innermost dimension, it is not able to capture the regularity, and thus compress, along outer dimensions. Hence, to be as efficient as our folding algorithm on our `backprop` example, for most execution instances, k would have to be bigger than 10^4. The complexity of the affine recognition algorithm of Rodriguez et al. [28] increases exponentially with the number of irregularities. So in practice, it has to give up even for nearly affine traces.

Similarly to us, existing runtime polyhedral optimizers [29, 25] use runtime information to create a polyhedral representation of a program. PolyJIT [29] focuses on handling programs that do not fit the polyhedral model statically because of memory accesses, loop bounds and conditionals that are described by quadratic functions involving parameters. Apollo [25] handle this case and many others preventing static polyhedral optimizers from operating. Compared to our analysis, PolyJIT focuses on identifying 100% affine programs which may be rare in practice for many reasons such as the memory allocation concerns pointed out for `backprop`. Apollo proposes a *tube* mechanisms [30] which allows the handling of programs with quasi-affine memory accesses. Even with this last extension, on the illustrative example of `backprop`, Apollo will, as opposed to our analysis, neither manage to over-approximate the non constant stride along the inner-most dimension as soon as the stride distance is greater than a given threshold, nor detect the stride of 1 along the outermost dimension. Also, it is worth mentioning that, as for the example of `backprop`, a program might show affine dependencies while having non-affine memory accesses. Contrary to our analysis front-end, which tracks both separately, Apollo only traces memory accesses and then recomputes the dependencies from them. Consequently, Apollo has to give up completely here while we can detect an accurate polyhedral representation of dependencies.

7 Conclusion and Perspectives

We have presented a folding algorithm able to create a polyhedral representation of a program from its execution trace. Based on a geometric approach, our algorithm scales to real-life applications by safely over-approximating the dependencies that do not fit the polyhedral model while still recovering precise information for those that do. From what we observed on the Rodinia benchmark, large regions of programs that are perfectly affine and suitable as-is for classic polyhedral optimizers are rare. Thanks to our over-approximation mechanisms, we are nevertheless still able to create a polyhedral representation for these programs that can be given to a classic polyhedral optimizer.

Regarding the perspectives opened by this work, we are already working in two directions that will allow handling more programs. The first one consists of adding new dimensions not present in the program to our representation. Said differently, an instruction contained in a 2-dimensional loop nest in the program could be represented by a 3-dimensional polyhedron. This mechanism, already at work in trace compression algorithms [19, 28] will allow our analysis to handle tiled stencil computations and programs where 2-dimensional arrays are traversed by linearized 1-dimensional loops. The second extension we want to investigate is a clever mechanism for the activation of widening for dependency label expressions. We are planning to replace the existing user controlled global option with an adaptive mechanism that automatically activates widening

as *needed*. For example, the option could be activated when the number of polyhedra used to represent a given instruction or dependency is becoming too large. This would allow having a trade-off between the accuracy and the size of the output of the folding algorithm.

References

[1] Ao, R., Tan, G., and Chen, M. Parainsight: An assistant for quantitatively analyzing multi-granularity parallel region. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on* (2013), IEEE, pp. 698–707.

[2] Bastoul, C. Generating loops for scanning polyhedra: Cloog users guide. *Polyhedron 2* (2004), 10.

[3] Bellard, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2005), ATEC '05.

[4] Berg, E., and Hagersten, E. Fast data-locality profiling of native execution. In *ACM SIGMETRICS Performance Evaluation Review* (2005), vol. 33, ACM, pp. 169–180.

[5] Beyls, K., and D'Hollander, E. Discovery of locality-improving refactorings by reuse path analysis. *High Performance Computing and Communications* (2006), 220–229.

[6] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral program optimization system. In *PLDI* (2008).

[7] Boulet, P., Darte, A., Silber, G.-A., and Vivien, F. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Comput. 24*, 3-4 (May 1998), 421–444.

[8] Butt, K., Qadeer, A., Mustafa, G., and Waheed, A. Runtime analysis of application binaries for function level parallelism potential using qemu. In *Open Source Systems and Technologies (ICOSST), 2012 International Conference on* (2012), IEEE, pp. 33–39.

[9] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009).

[10] Che, S., Sheaffer, J. W., Boyer, M., Szafaryn, L. G., Wang, L., and Skadron, K. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)* (Washington, DC, USA, 2010), IISWC '10, IEEE Computer Society, pp. 1–11.

[11] Collard, J.-F., Barthou, D., and Feautrier, P. Fuzzy array dataflow analysis. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1995), PPOPP '95, ACM, pp. 92–101.

[12] Faxén, K.-F., Popov, K., Jansson, S., and Albertsson, L. Embla - data dependence profiling for parallel programming. In *Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems* (Washington, DC, USA, 2008), CISIS '08, IEEE Computer Society, pp. 780–785.

[13] Feautrier, P. Parametric integer programming. *RAIRO-Operations Research 22*, 3 (1988), 243–268.

[14] Feautrier, P., and Lengauer, C. Polyhedron model. In *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1581–1592.

[15] Grosser, T., Groesslinger, A., and Lengauer, C. Polly - performing polyhedral pptimizations on a low-level intermediate representation. *Parallel Processing Letters 22*, 04 (2012), 1250010.

[16] Gruber, F., Selva, M., Sampaio, D., Guillon, C., Moynault, A., Pouchet, L.-N., and Rastello, F. Data-flow/dependence profiling for structured transformations. In *Submitted to the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)* (Feb. 2019).

[17] Guillon, C. Program instrumentation with qemu. In *Proceedings of the International QEMU User's Forum* (2011), QUF '11.

[18] Holewinski, J., Ramamurthi, R., Ravishankar, M., Fauzia, N., Pouchet, L.-N., Rountev, A., and Sadayappan, P. Dynamic trace-based analysis of vectorization potential of applications. *ACM SIGPLAN Notices 47*, 6 (2012), 371–382.

[19] Ketterlin, A., and Clauss, P. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2008), CGO '08, ACM, pp. 94–103.

[20] Ketterlin, A., and Clauss, P. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2012), MICRO-45, IEEE Computer Society, pp. 437–448.

[21] Kim, M., Kim, H., and Luk, C.-K. Prospector: A dynamic data-dependence profiler to help parallel programming. In *HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism* (2010).

[22] Li, Z., Atre, R., Ul-Huda, Z., Jannesari, A., and Wolf, F. Discopop: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014*. Springer, 2015, pp. 37–54.

[23] Liu, X., and Mellor-Crummey, J. Pinpointing data locality problems using data-centric analysis. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on* (2011), IEEE, pp. 171–180.

[24] Marin, G., Dongarra, J., and Terpstra, D. Miami: A framework for application performance diagnosis. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (March 2014).

[25] Martinez Caamaño, J. M., Selva, M., Clauss, P., Baloian, A., and Wolff, W. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience 29*, 15 (2017), e4192. e4192 cpe.4192.

[26] Nethercote, N., and Mycroft, A. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science 89*, 2 (2003), 149–170.

[27] Pop, S., Cohen, A., and Silber, G.-A. Induction variable analysis with delayed abstractions. In *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers* (2005), HiPEAC'05.

[28] Rodríguez, G., Andión, J. M., Kandemir, M. T., and Touriño, J. Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (New York, NY, USA, 2016), CGO '16, ACM, pp. 139–149.

[29] Simbürger, A., Apel, S., Grösslinger, A., and Lengauer, C. Polyjit: Polyhedral optimization just in time. *International Journal of Parallel Programming* (Aug 2018).

[30] Sukumaran-Rajam, A., and Clauss, P. The polyhedral model of nonlinear loops. *ACM Trans. Archit. Code Optim. 12*, 4 (Dec. 2015), 48:1–48:27.

[31] Tournavitis, G., and Franke, B. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 377–388.

[32] Trifunovic, K., Cohen, A., Edelsohn, D., Li, F., Grosser, T., Jagasia, H., Ladelsky, R., Pop, S., Sjödin, J., and Upadrasta, R. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)* (Pisa, Italy, Jan. 2010), ACM.

[33] Van Engelen, R. A. Efficient symbolic analysis for optimizing compilers. In *International Conference on Compiler Construction* (2001), Springer.

[34] Vandierendonck, H., Rul, S., and De Bosschere, K. The paralax infrastructure: automatic parallelization with a helping hand. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on* (New York, NY, USA, 2010), ACM, pp. 389–399.

[35] Wang, Z., Tournavitis, G., Franke, B., and O'boyle, M. F. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization (TACO) 11*, 1 (2014), 2.

```
# Per dimension list of absorber polyhedra.
<int, poly_list_t> absorbers

# Per dimension dictionary mapping
# extreme points to polyhedra to be absorbed
<int, <point_t, poly_t>> eps_2_to_be_absorbed

# While we have points
while(True):
  pt = wait_next_point()
  if point == end_of_stream: break

  # Put current point in absorbers[0]
  absorbers[0].insert(new Polyhedron(point))

  # for each dimension d such that d=1 or loop d-1 completed
  for d in process_dims(pt):

    # Step 1: promote absorbers[d-1] -> eps_2_to_be_absorbed[d]
    for p in absorbers[d-1]:
      p.move(absorbers[d-1], eps_2_to_be_absorbed[d])

    # Step 2: absorbers[d] try to absorb eps_2_to_be_absorbed[d]
    for abso in absorbers[d]:
      absorbed = False
      for v in abso.search_vectors:
        corner    = abso.upper_left
        to_be_abs = eps_2_to_be_absorbed[d][corner + v]
        if to_be_abs != None:
          if (abso.can_absorb(to_be_abs, d) and
              abso.has_compat_label(to_be_abs, d)):
            abso.absorbs(to_be_abs, d)
            absorbed = True
            break

      if not absorbed:
        # abso will never absorb anyone along d,
        # then promote it in the next dimension
        abso.move(absorbers[d], eps_2_to_be_absorbed[d+1])

    # Step 3: promote all of remaining
    # eps_2_to_be_absorbed[d] -> absorbers[d]
    for not_abs in eps_2_to_be_absorbed[d].values:
      not_abs.move(eps_to_2_to_be_absorbed[d], absorbers[d])

# Stream finished, flush all pending polyhedra
flush_pending_polyhedra()
```

Algorithm 1: The main folding algorithm

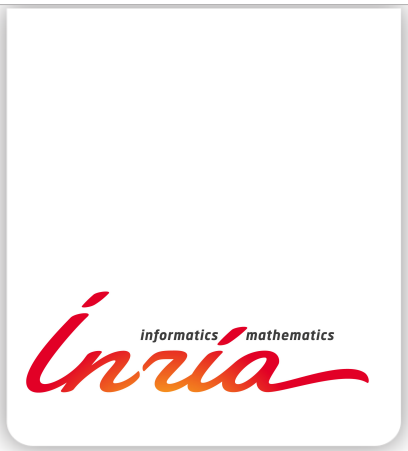
```
# Update coefficient for dimension d and remaining
# value of f_abs. No need to update f_to_be_abs
# because it will be thrown after absorption
def update_label_dims_known(f_abs, f_to_be_abs, d):
  # Update of coefficient
  new_coeff = f_to_be_abs.coeffs[0] - f_abs.coeffs[0]
  f_abs.coeffs[d] = new_coeff
  # Update of remaining value
  new_coeff_contrib = new_coeff * f_abs.init_point[d]
  f_abs.coeffs[0] = f_abs.coeffs[0] - new_coeff_contrib

# Check in case absorber already has a coeff for d
def has_compat_label_dims_known(f_abs, f_to_be_abs, d):
  new_coeff_contrib = f_abs.coeffs[d] * f_to_be_abs.coeffs[d]
  new_remain = f_to_be_abs.coeffs[0] - new_coeff_contrib
  return new_remain == f_abs.coeffs[0]
```

Algorithm 2: Update and compatibility check when all dimensions below d are known

```
def has_compat_label_general(f_abs, f_to_be_abs, d):
  if f_abs.coeff_types[d] == ⊥:
    return True
  abs_diff = 0
  to_be_abs_diff = 0
  for q in [1, d]:
    abs_t = f_abs.coeff_types[q]
    to_be_abs_t = f_to_be_abs.coeff_types[q]
    if abs_t != ⊥ and to_be_abs_t != ⊥:
      if f_abs.coeffs[q] != f_to_be_abs.coeffs[q]:
        return False
    if abs_t == ⊥ and to_be_abs_t != ⊥:
      abs_diff += f_to_be_abs.coeffs[q] * f_abs.init_point[d]
    if abs_t != ⊥ and to_be_abs_t == ⊥:
      to_be_abs_diff += f_abs.coeffs[q]*f_to_be_abs.init_point[d]
  return f_abs.coeff[0] - abs_diff ==
     f_to_be_abs.coeff[0] - to_be_abs_diff
```

Algorithm 3: General compatibility check



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399