Building a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of Nonaffine Programs Scalable

MANUEL SELVA, FABIAN GRUBER, and DIOGO SAMPAIO, University of Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
CHRISTOPHE GUILLON, STMicroelectronics
LOUIS-NOËL POUCHET, Colorado State University
FABRICE RASTELLO, Univeristy of Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

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 the *folding-based analysis* that, from the output of an instrumented program execution, builds a compact polyhedral representation. It is able to accurately detect affine dependencies, fixed-stride memory accesses, and induction variables in programs. It scales to real-life applications, which often include some nonaffine dependencies and accesses in otherwise affine code. This is enabled by a safe fine-grained polyhedral overapproximation mechanism. We evaluate our analysis on the entire Rodinia benchmark suite, enabling accurate feedback about the potential for complex polyhedral transformations.

CCS Concepts: • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Performance feedback, polyhedral model, loop transformations, compiler optimization, binary, instrumentation, dynamic dependency graph

ACM Reference format:
Manuel Selva, Fabian Gruber, Diogo Sampaio, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello. 2019. Building a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of Nonaffine Programs Scalable. *ACM Trans. Archit. Code Optim.* 16, 4, Article 45 (December 2019), 26 pages.
https://doi.org/10.1145/3363785

1 INTRODUCTION

The most effective program optimizations for improving performance or energy consumption are typically based on rescheduling of instructions so as to expose data locality and/or parallelism.

M. Selva, F. Gruber, and D. Sampaio contributed equally to this research.
This work was supported in part by the U.S. National Science Foundation awards CCF-1645514 and CCF-1750399, and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.
Authors' addresses: M. Selva, F. Gruber, and D. Sampaio, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG; emails: {manuel.selva, fabian.gruber, diogo.sampaio}@inria.fr; C. Guillon, STMicroelectronics; email: christophe.guillon@st.com; L.-N. Pouchet, Colorado State University; email: pouchet@colostate.edu; F. Rastello, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG; email: fabrice.rastello@inria.fr.

1544-3566/2019/12-ART45
https://doi.org/10.1145/3363785

Instruction rescheduling techniques range from basic block reordering to multiloop transformations such as vectorization or loop nest tiling [7]. The detection of parallelism and spatial locality properties along existing loops typically does not need a sophisticated representation of the data dependencies themselves, as validating the absence of dependencies or computing the dependence distance is usually sufficient. On the other hand, assessing the applicability of loop *transformations* that implement complex instruction rescheduling such as loop skewing, interchange, or tiling requires precisely characterizing the dependencies. The polyhedral model [16] is an appropriate representation for this purpose. Bondhugula et al. demonstrated how tiling applicability can be increased using polyhedral dependencies and scheduling [7]. Polyhedral optimizers [7, 17, 34, 42, 45] 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-grained (i.e., SIMD) parallelism.

Dynamic dependence analysis has been shown to be a useful tool for finding optimization potential. Most existing techniques allow to efficiently inform about the absence of dependencies (for the particular executions of the program that have been instrumented [14]) along loops in the original program, highlighting opportunities for parallelism [25, 41, 44, 46] or SIMD vectorization [23]. Building a polyhedral representation of dependencies and memory access patterns with dynamic information is a natural way to detect whether applying transformations such as skewing or loop permutation is legal. Unfortunately, the practical development of such a dynamic analysis faces a number of problems. In particular, supporting applications that are not fully affine is challenging. Existing solutions either use overly pessimistic approximations and lose information [40] or do not scale well to large problem sizes [24, 35].

In this work, we propose a dynamic analysis, called the *folding-based analysis*, that addresses these challenges. We successfully used it as a cornerstone of a complete profiling tool chain [19]. This tool chain works on compiled binaries and provides suggestions for loop transformations. It then uses debugging data to map the suggestions back to the source code to guide users to where they should apply these transformations. The tool chain consists of three parts:

- The *front-end*, which instruments the binary to get information from a program execution
- The *folding-based analysis*, which consumes this information in a streaming fashion to build a compact polyhedral program representation
- The *back-end*, which uses this representation to find and suggest interesting loop transformations

The complete tool chain has been demonstrated in prior work [19], focusing on the overall design and applicability of our approach. The present article details the underlying techniques in folding-based analysis, which is a cornerstone of the profiling tool chain. This article is an updated and peer-reviewed version of our prior technical report [20].

The folding-based analysis can accurately detect polyhedral dependencies in programs and scales to real-life applications, which often include some nonaffine dependencies in otherwise affine code. For that, we propose a safe fine-grained polyhedral overapproximation mechanism for such dependencies. That is, our analysis emits a compact program representation allowing a classic polyhedral optimizer to find a wide range of possible transformations. Our analysis also allows detecting the presence of fixed-stride memory accesses and induction variables. Fixed-stride memory accesses are useful for exposing the potential for vectorization and loop transformations that improve spatial locality. Detecting induction variables allows removing unnecessary dependencies.

```
1  for (j = 1; j <= n2; j++) { // For each unit in second layer
2    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);
6  }
```

Fig. 1. A compute-intensive kernel in backprop.

The contribution of this article is the folding-based analysis, which:

- scales to real-life applications thanks to a safe polyhedral overapproximation mechanism applied to nonaffine parts of the program;
- builds a compact polyhedral program representation from a program execution, enabling polyhedral optimizations to be applied, that is, to provide feedback about the potential of complex polyhedral transformations; and
- captures information useful for polyhedral optimizers such as properties of dataflow dependencies, memory accesses, and induction variables in a uniform manner.

An open source implementation of the folding-based analysis is available at [18].

This article is organized as follows. Section 2 illustrates the context of our work through a case study. Section 3 discusses related work. Section 4 then continues with an in-depth description of the interface, followed by the core algorithm used by our analysis in Section 5. Section 6 evaluates our approach by applying it to the entire Rodinia benchmark suite [11]. Finally, Section 7 concludes the article and provides future perspectives.

2 ILLUSTRATIVE 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 [11]. `backprop` is a supervised learning method used to train artificial neural networks. We focus on the compute kernel shown in Figure 1. This kernel is also used as a running example throughout the rest of the article. For complex real-life case studies, the reader should refer to our work describing the entire profiling tool [19].

2.1 Example Problem: `backprop`

The main source of inefficiency in `backprop` is the 2D access to `conn` on Line 4. The problem here is that `conn` is laid out in row-major order, but the accesses are in column-major order. This leads to unnecessary cache misses. A loop interchange, which switches the order of the `j` and the `k` loop, solves this problem and furthermore unlocks vectorization opportunities. Identifying the profitability of these transformations requires detecting the strided access along the outer `j` loop.

However, reconstructing this striding information from the stream of memory addresses being accessed in a dynamic analysis is not trivial. This is because `conn` is not a 2-dimensional array, but an array of pointers, each allocated by a separate call to `malloc` as illustrated in Figure 2. Since `malloc` gives no guarantees on the placement of allocations, the accesses along the innermost `k` dimension do not have a constant stride but are irregular as shown in Figure 3.

Existing dynamic polyhedral approaches [24, 35] try to build a completely affine model and are not able to handle even only partially irregular applications like `backprop`. These algorithms do not take iterator values into account and directly work on a linear stream of memory addresses. The irregularity along the inner `k` loop either stops them from detecting that there even is an outer `j` or causes them to exhibit a prohibitively high time and space complexity. There is an approach that takes iterator values into account, which allows them to tolerate some irregular accesses using

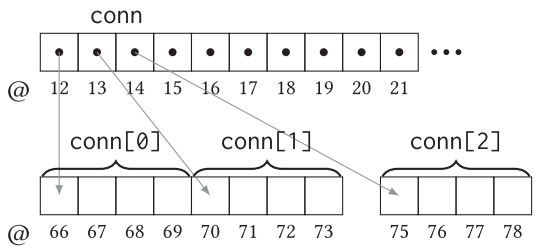

Fig. 2. Memory layout for the conn array with n2 = 3. For simplicity, pointers and numbers fit in 1 byte.

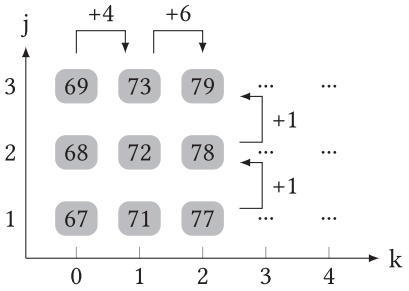

Fig. 3. Addresses used to access conn[k][j].

approximation [30, 40]. However, this approximation mechanism is relatively conservative, and if two cells of the conn are too far apart in memory, it will completely give up on trying to model the loop nest. As a consequence, none of the existing dynamic approaches scale beyond very small input datasets.

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 contiguously in memory even if malloc gives no guarantees on the placement of allocations. In this case the memory accesses for conn[k][j] are regular along both k and j, and existing dynamic polyhedral approaches are able to model and optimize this synthetic benchmark. However, other more realistic applications such as, for example, sparse matrix algorithms inherently exhibit irregular access patterns, the same as those in backprop [39, Section 5.2].

2.2 Solution: Folding-based Analysis

Despite the lack of information about aliasing and the presence of a nonaffine 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 and
- the absence of dependence along dimension j.

From this information, our back-end was able to suggest loop interchange, vectorization, and tiling, which in our case led to a speedup of ×5.3 [19].

The vectorization opportunity is revealed by looking at the scalar evolution [33, 43] 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 Figure 2, can be described with the function $0j + 1k + 12$, 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 function. This is shown in Figure 3. Our analysis is robust against this irregularity along dimension k and is able to produce the incomplete function $1j + \top k + 66$, 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 function does indicate that the memory address increases by one every iteration of dimension j. We refer to this as a *stride-1* access.

The folding algorithm discovers not only the structure of memory accesses but also the structure of data dependencies in general. In our running example, it detects that there is no dependence between the reads on Line 4 and the write on Line 5. It is worth mentioning that while the

algorithm is exactly the same for both, the structure of memory accesses and dependencies are detected separately. The folding algorithm can thus handle cases where accesses are nonaffine and dependencies are affine, and vice versa. Here the irregularity of the former does not hinder the folding algorithm from finding the structure of the latter.

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. Our analysis, like any dynamic approach reasoning on an execution, cannot guarantee that this holds in general, but it can still provide useful feedback. 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.

3 RELATED WORK

Integer linear algebra is a natural formalism for representing the computation space of a loop nest. The polyhedral framework [16] leverages, among others, operators on polyhedra, parametric integer linear programming [15] for dependence analysis [12], and enumeration for code generation [3]. 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 [42] and LLVM-Polly [17]. 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 [7].

Dynamic data dependency analysis is a technique typically used to provide feedback to the programmer, e.g., about the existence or absence of dependencies along loops. The detection of parallelism along canonical directions, such as vectorization, has been particularly investigated [2, 9, 14, 25–27, 41, 44, 46], as it requires only relatively localized information. Another use case is the evaluation of effective reuse [5, 6, 28, 29] with the objective of pinpointing data locality problems. Like us, with the objective of gathering more global dependency information, Redux [31] builds a complete extended dynamic dependency graph from binary level programs. The article concludes with a negative result. Because of its inability to compress the produced graph, it is only able to handle very small nonrealistic programs.

Existing trace compression algorithms [24, 35] can be used to extract a polyhedral representation from an instrumented program execution. However, although they excel in rebuilding a polyhedral representation for a purely affine execution, they suffer inherent limitations for even partially nonaffine traces. They share the idea of using pattern matching with affine functions with our folding algorithm but do not exploit geometric information like we do. The nested loop recognition algorithm of Ketterlin et al. [24] detects outer loops by maintaining and repeatedly examining a finite window of memory accesses. Another algorithm by Rodriguez et al. [35] instead introduces a new loop into its representation every time it cannot handle an access. For perfectly regular programs, Ketterlin's approach only requires a small window and Rodriguez's will only create as many loops as there are in the original program. In that simple case, using a geometric approach does not make much difference and both algorithms are very efficient. That is, for regular programs, with D the dimension of the iteration space and n the number of points, the complexity of both nongeometric approaches is $O(n)$, the same as our approach. However, in the context of profiling large nonfully affine programs, neither of these two existing approaches can be used. The complexity of Ketterlin's algorithm increases quadratically with a parameter k that bounds the size of the window. Unfortunately, this forces a tradeoff between speed and quality of the output when choosing the size of this 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. On the other hand, the complexity of Rodriguez's approach increases exponentially with the number of irregularities. So in practice, it has to give up even for nearly affine traces.

PSnAP [32] is a memory access trace compression system for performance modeling and trace-based simulation of scientific applications. Its compression is based on the dynamic detection of the frequency at which strides occur along the innermost loop dimension.

Streaming convex hull algorithms [8, 22] could also be applied to build a compact geometric representation of an instrumented execution. However, our approach is able to precisely represent nonconvex polyhedra via a union of convex ones, while convex hull can only approximate this case with a single polyhedron.

Similarly to us, existing runtime polyhedral optimizers [30, 38] use runtime information to create a polyhedral representation of a program. PolyJIT [38] 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 [30] handles this case and many others preventing static polyhedral optimizers from operating. Compared to our analysis, PolyJIT focuses on identifying 100% of affine programs that may be rare in practice for many reasons such as the memory allocation concerns pointed out for `backprop`. Apollo proposes a *tube* mechanism [40] that allows the handling of programs with quasi-affine memory accesses. Because Apollo focuses on implementing polyhedral transformations at runtime automatically, the overapproximation performed by the tube mechanism has to be efficiently verified during the execution. Indeed, that the approximation is safe has been verified along with every execution of the optimized code. Hence, the check has to be simple so as to have a very low overhead. This is not the case in our context, since we only provide transformation suggestions to the programmer. As a consequence, even with the tube extension, on the illustrative example of `backprop`, Apollo will, as opposed to our analysis, neither manage to overapproximate the nonconstant stride along the innermost 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 nonaffine 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.

Since the transformations proposed by our back-end are based on information gathered during one program execution, they are not guaranteed to always be valid. Hybrid analyses using code versioning combined with runtime alias checks [1, 13] or dynamic checks for the validity of transformations [36, 37] can optimize programs even when the optimizations are not guaranteed to always be legal. However, these approaches still have the problem of determining if applying a transformation would be profitable. Profiling for profitability such as done by POLY-PROF could be integrated to guide these hybrid systems to decide what parts of a program to optimize.

4 INTERFACE OF THE FOLDING-BASED ANALYSIS

Before describing the core algorithm of the folding-based analysis in Section 5, we introduce its inputs and outputs.

4.1 Inputs

The front-end of POLY-PROF, implemented using the dynamic binary translator QEMU [4, 21], instruments programs to produce the input for the folding-based analysis as they execute. The trace generated by the instrumentation is then directly processed by the folding-based analysis as the program executes. 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. Even

```
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
```

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

though the front-end analyzes machine code, it works at the level of the generic QEMU IR, making it CPU architecture agnostic.

The inputs of the folding algorithm are streams of two types, one for instructions and one for dependencies. In the following sections, a *static instruction* is a machine instruction in the program binary. An *instruction instance* is one dynamic execution of a static instruction. A dependency is a pair consisting of an instruction instance that produced a value and another instance consuming it. Also, our front-end only captures dataflow dependencies, that is, read-after-write dependencies for which there are no intermediate writes to the same memory location or register.

Each input stream has a unique *identifier Id*. An instruction stream is identified by a static instruction. A stream of data dependencies is identified by a pair of static instructions. We note this as *Static instruction source* → *Static instruction destination*. 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, all the *IV*s span a multidimensional space where each entry is a point. Thus, in the following we use the terms *entry* and *point* interchangeably. Also, note that *IV*s arrive in the input stream in lexicographical order.

Finally, it is worth mentioning that the front-end tracks the calling context in which instructions execute and generates different input streams for different calls to the same function [19].

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 dealing with integers, the label is the integer value representing the result computed by the instruction. If the instruction is a memory access, the label is the address read or written by the instance. As described in the next section, these labels are used to identify induction variables and fixed-stride memory accesses.

To illustrate the contents of the input stream of instruction instances, we again use the example of `backprop` from Figure 1. At the binary level, the considered loop-nest contains several instructions that are represented in an abstract C-like fashion in Figure 4. An excerpt of the six 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 dependency stream for a pair of static instructions 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 the destination, whereas the label is the IV of the source. Table 2 shows three of the six dependency

Table 1. Instruction Input Streams from Example in Figure 4 with n1 = 42

Id = I1		Id = I2		Id = I3		Id = I4		Id = I5		Id = I6	
IV (cj, ck)	Label	*IV* (cj, ck)	Label	*IV* (cj, ck)	Label	*IV* (cj, ck)	Label	*IV* (cj, ck)	Label	*IV* (cj)	Label
(0, 0)	12	(0, 0)	67	(0, 0)	407	(0, 0)	N/A	(0, 0)	1		
(0, 1)	13	(0, 1)	71	(0, 1)	408	(0, 1)	N/A	(0, 1)	2		
...	...	...	...	...	...	...	...	...	...		
(0, 42)	54	(0, 42)	243	(0, 42)	449	(0, 42)	N/A	(0, 42)	43	(0)	2
(1, 0)	12	(1, 0)	68	(1, 0)	407	(1, 0)	N/A	(1, 0)	1		
(1, 1)	13	(1, 1)	72	(1, 1)	408	(1, 1)	N/A	(1, 1)	2		
...	...	...	...	...	...	...	...	...	...	...	...

Table 2. Three of the Six Dependency Input Streams from Example in Figure 4

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)
...	...	...	...	...	...

input streams for the example in Figure 4. In this example, all the dependencies except I4 → I4 are intra-iteration dependencies.

4.2 Outputs

The folding algorithm processes each stream independently. For each stream, the final result of folding is a piecewise linear function mapping *IVs* to labels. We refer to this piecewise linear function as a *label function*. The domain of a label function contains exactly the *IVs* of all entries of the input stream. Moreover, when the label function is applied to an *IV* of its domain, it produces the label value associated with that point in the input stream. That is, a label function is a compact representation of an 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.

Each piece of the domain of a label function is described by a set of affine inequalities, and hence it defines a *polyhedron*. More precisely, a label function can be written as:

$$f : \mathbb{N}^d \mapsto \mathbb{Z} \mid f(c_1, \ldots, c_d) = \begin{cases} k_{0,1} + \sum_{1 \le i \le d} k_{i,1} c_i & \text{if } (c_1, \ldots, c_d) \in \text{polyhedron}_1 \\ k_{0,2} + \sum_{1 \le i \le d} k_{i,2} c_i & \text{if } (c_1, \ldots, c_d) \in \text{polyhedron}_2 \\ \ldots, \end{cases}$$

where, $k_{0,j} \in \mathbb{Z}$, $\forall i \ge 1 \; . \; k_{i,j} \in \mathbb{Z} \cup \{\top\}$, multiplication with $\top$ is defined as $\top c_i = U_{i,j}(c_i)c_i$, and $U_{i,j}$ is an uninterpreted function of c_i.

The coefficients of a label function may be either an integer or an uninterpreted function $U_{i,j}$ represented as $\top$[1]. The use of an uninterpreted function as a coefficient indicates that the evolution of the label cannot compactly be expressed as an affine function along the corresponding

[1]As described later, the folding algorithm internally also uses special $\bot$ values for coefficients that have not yet been determined, but these do not appear in the output.

Table 3. Output of the Folding Algorithm for the Instructions Stream of Table 1 with n2 = 16 and n1 = 42

Id	Polyhedron (cj, ck)	Label function f(cj, ck)
I1	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$0cj + 1ck + 12$
I2	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$1cj + \top ck + 67$
I3	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$0cj + 1ck + 407$
I4	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	N/A
I5	$0 \leq cj \leq 15, 0 \leq ck \leq 42$	$0cj + 1ck + 1$
I6	$0 \leq cj \leq 15$	$1cj + 2$

dimension. This occurs, for example, when modeling the addresses accessed by the nonaffine load I2 in `backprop` shown in Figure 3.

When a label function does not contain any $\top$ coefficient, it can be used to precisely reconstruct the input stream it was created from. As mentioned before, the domain of a label function contains all *IVs* from the input and no other points. We can thus reconstruct the stream simply by applying the label function to every point of its domain.

If a label function does contain $\top$, one can no longer apply it to a point to produce a value, since it is defined by an uninterpreted function $U_{i,j}$, but the remaining non-$\top$ coefficients still allow reasoning about the values seen in the input. For example, in the function $f(j,k) = 1j + \top k + 66$ from Section 2.2, we know that $f(j,k) - f(j',k) == j - j'$.

Instructions. For an instruction stream, depending on the type of its corresponding static instruction, the label function represents either the integer values computed by the instruction or the addresses it accesses. These label functions are then used to identify induction variables and fixed-stride memory accesses. Table 3 illustrates the outputs for the input streams in Algorithm 1, where `n2 = 16` and `n1 = 42`. All instruction instances of a given input stream are now described by a single line. We notice from this table that four of the six instructions have an affine function where all the coefficients are known; that is, they are not $\top$. The affine function of instruction I4 is marked as N/A because it is computing floating-point values. Instruction I2 has an affine function with the coefficient for dimension k being $\top$, as already discussed. In this case, the labels of the input stream cannot be reconstructed from the *IVs*. Nevertheless, the algorithm still outputs the single polyhedron describing the domain for this instruction and produces useful information for a polyhedral optimizer. It is also worth mentioning that, unlike in this example, the label function of each instruction can be made up of several pieces if the domain of the instruction cannot be represented as a single convex polyhedron. In this case, the domain would be represented as a union of polyhedra.

Dependencies. The label function of a dependency is a piecewise linear function with multiple outputs. The label function maps *IVs* of the consumer instances of the dependence to *IVs* of the producer instances. That is, given an instruction instance, the label function can be used to determine from which other instruction instance 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

Table 4. Output of the Folding Algorithm for the Dependencies Stream Shown in Table 2

Id	Polyhedron (cj,ck)	Label function 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	$0 \leq cj \leq 15, 1 \leq ck \leq 42$	$cj' = cj + 0ck, ck' = 0cj + ck - 1$

dependencies is described by a single polyhedron. Nevertheless, in more complex cases these domains can be represented by a union of polyhedra.

4.3 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 rescheduling 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 function where all coefficients of all pieces are integers, that is, not $\top$. The initial loop iterators are an example of induction variable, that is, 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 function. 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, the complexity of which is combinatorial with the size of the polyhedral representation. To that end, we overapproximate 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 overapproximated dependency given to the back-end would be $cj' = cj \wedge ck' \leq ck$.

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 function of an instruction accessing memory.

5 THE FOLDING ALGORITHM

This section gives an overview of the folding algorithm and then presents its components in detail.

5.1 Overview

As stated in the previous section, the folding algorithm processes the stream for each identifier separately. It is worth mentioning that exactly the same algorithm is used for both instruction and

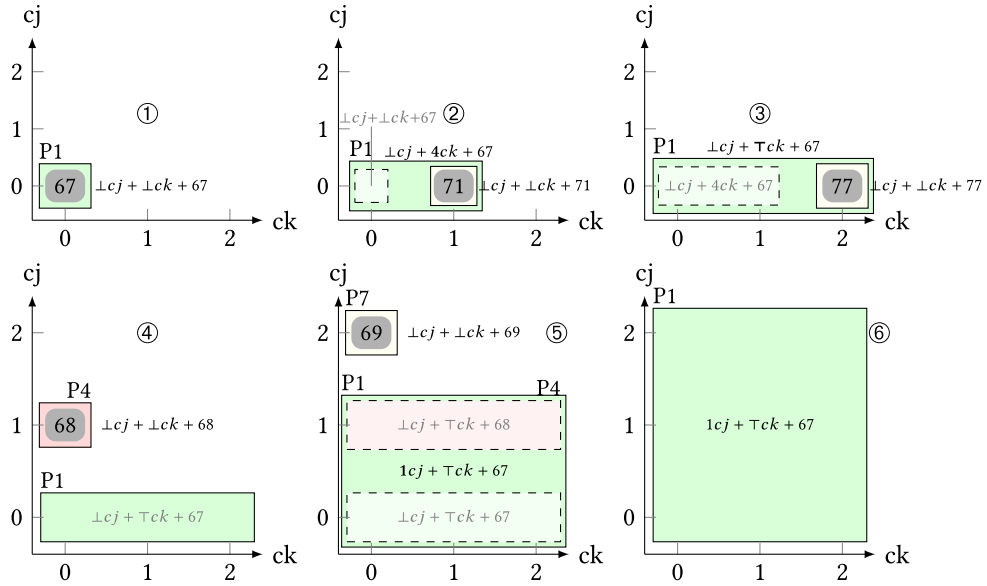

Fig. 5. Folding process for the input stream of I2 in Table 1 considering only three points in both dimension.

dependency streams. This algorithm receives points in a geometrical space as specified by the *IVs*. The main idea of the algorithm is to construct polyhedra from those points. For each polyhedron the algorithm also constructs an affine function 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.

5.1.1 Geometric Folding. The folding process for I2 is illustrated in Figure 5. For now we will ignore the construction of the affine function. 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 Figure 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 Figure 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 Figure 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.

5.1.2 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 Figure 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 $\perp cj + 4ck + 67$ when P1 absorbs the received point $(cj = 0, ck = 1)$ with label 71 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 Figure 5).
⑤ Update $f1$ to $f1(cj, ck) = 1cj + \top ck + 67$ when P1 absorbs P4 because cj is 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.

When the folding algorithm finishes, all remaining $\perp$ coefficients can safely be set to zero. The intuition behind this is that at the end of the folding process, a $\perp$ coefficient signals that the loop for this dimension only iterated once; that is, it never influenced the label value.

The algorithm that folds the labels of a dependency is the same as the one described above for the labels 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.

5.2 The Algorithm

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

5.2.1 Main Folding Function. The main function is shown in Algorithm 1. As explained in Section 4, 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 9). 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 Figure 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.

Elementary Polyhedra. The absorption process, explained in Section 5.2.2, is restricted to only produce a subclass of convex polyhedra that we call *elementary polyhedra*. Because the folding algorithm only produces elementary polyhedra, the term *polyhedra* implicitly refers to elementary polyhedra in the following. A d-dimensional elementary polyhedron is a convex polyhedron with 2^d vertices and a restricted shape. The shape restriction is motivated by complexity concerns for the absorption process as explained in Section 5.3.

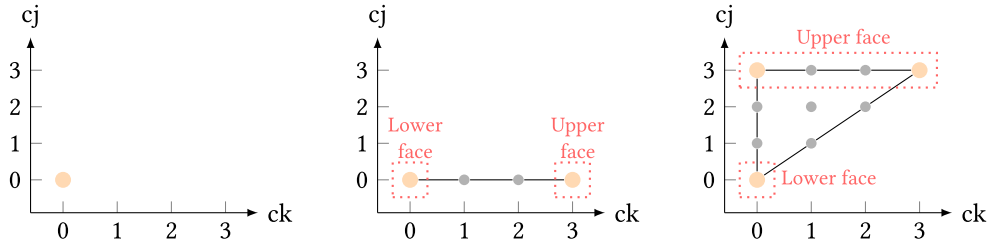

Fig. 6. Examples of elementary polyhedra in a 2-dimensional space (D = 2) with its vertices in orange.

We define elementary polyhedra in a D-dimensional space 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. In other words, the polyhedron is flat on dimensions between $d + 1$ and D;
  (2) it has two $(d - 1)$-faces flat on dimension d but with different coordinates for that dth dimension. The face with the lower coordinates in d is called the *lower face* and the one with the higher coordinates is called the *upper face*;
  (3) its lower and upper faces must themselves be $(d - 1)$-elementary polyhedra;
  (4) 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 $+1$.

All faces of an elementary polyhedron beside the upper and lower face are called *side faces*.

More informally, an elementary 0-dimensional polyhedron is a polyhedron made of a single point. An elementary 1-dimensional polyhedron is an interval. An elementary 2-dimensional polyhedron is a trapezoid. An elementary 3-dimensional polyhedron is a trapezoidal prism. Every general polyhedron can be represented using unions of elementary polyhedra, meaning any iteration or dependence space can be described with them. The more regular a space is, the fewer elementary polyhedra are necessary to represent it.

A polyhedron is *degenerate* on a given dimension if all its vertices 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.

Figure 6 shows examples of elementary polyhedra in a 2-dimensional space (D = 2). The vertices of the polyhedra are shown as large dots. The other integer points included in the polyhedra are shown with small dots. Note that even though the lower face in Figure 6(c) is degenerate, it is still represented using two vertices, but they have the same coordinates.

Producing only elementary polyhedra as described above allows controlling the worst-case complexity of the absorption process described in Section 5.2.2. 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 canonical *IVs* starting at zero and only ever advancing by one. Hence, elementary polyhedra are able to represent the iteration space of most of the loops fitting the polyhedral model.

ALGORITHM 1: The main folding algorithm

```
1  # Per dimension list of absorber polyhedra.
2  <int, poly_list_t> absorbers
3  # Per dimension dictionary mapping vertices to polyhedra to be absorbed
4  <int, <point_t, poly_t>> vertices_2_to_be_absorbed
5
6  # While we have points
7  while(True):
8    # End of stream?
9    point = wait_next_point()
10   if point == end_of_stream:
11     break
12   # Put current point in absorbers[0]
13   absorbers[0].insert(new Polyhedron(point))
14   # for each dimension d such that d=1 or loop d-1 completed
15   for d in process_dims(point):
16     # Step 1: promote absorbers[d-1] -> vertices_2_to_be_absorbed[d]
17     for p in absorbers[d-1]:
18       p.move(absorbers[d-1], vertices_2_to_be_absorbed[d])
19     # Step 2: absorbers[d] try to absorb vertices_2_to_be_absorbed[d]
20     for abso in absorbers[d]:
21       absorbed = False
22       for v in abso.search_vectors:
23         corner    = abso.upper_left
24         to_be_abs = vertices_2_to_be_absorbed[d][corner + v]
25         if to_be_abs != None:
26           if has_compat_label(abso, to_be_abs, d) and
27              has_compat_geometry(abso, to_be_abs, d):
28             update_geometry(abso, to_be_abs, d)
29             update_label(abso.label_function, to.label_function)
30             absorbed = True
31             break
32       if not absorbed:
33         # abso will never absorb anyone along d, then promote it in the next dimension
34         abso.move(absorbers[d], vertices_2_to_be_absorbed[d+1])
35     # Step 3: promote all of remaining vertices_2_to_be_absorbed[d] -> absorbers[d]
36     for not_abs in vertices_2_to_be_absorbed[d].values:
37       not_abs.move(vertices_2_to_be_absorbed[d], absorbers[d])
38 # Stream finished, flush all pending polyhedra
39 flush_pending_polyhedra()
```

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 stored using 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 `vertices_2_to_be_absorbed[d]` (Line 4). The keys of the dictionary `vertices_2_to_be_absorbed[d]` are the lexicographically smallest vertices of the polyhedra to be absorbed. This point, which we name the *anchor*, is used to uniquely identify the absorbed polyhedron. `abso.upper_left` (Line 23), the lexicographically smallest point of the upper face of `abso`, also called its *corner*, is the vertex from which the absorption is done. For example, in Figure 5, in the absorption just before step 6, the anchor of P7 is $(cj = 2, ck = 0)$ and `upper_left` of P1 is $(cj =, ck = 0)$.

Steps of the algorithm. 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 no loop finishes, the algorithm processes only dimension $d = 1$; if the innermost loop finishes, it processes dimensions $d = 1$ and $d = 2$; if its enclosing loop finishes, it processes dimensions $d = 1$, $d = 2$, and $d = 3$; and so forth. In Line 15, `process_dims` represents that set of dimensions to be processed.

Before processing the different dimensions, the current point is added into `absorbers[0]` (Line 13). This state is only transient, because as soon as the innermost dimension is processed, the point will be promoted into `vertices_2_to_be_absorbed[1]` (Line 18). Then, for each dimension d of `process_dims` (processed from inner to outer), three steps are performed.

The first step (Lines 17 to 18) promotes all polyhedra in `absorbers[d-1]` into `vertices_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]` has 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 20 to 34), polyhedra from `absorbers[d]` try to absorb polyhedra in `vertices_2_to_be_absorbed[d]`. For absorption to be possible, the polyhedra should be geometrically compatible (Line 27) and their label functions should match (Line 26) as described in Section 5.2.1 and Section 5.2.2. 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 `vertices_2_to_be_absorbed[d+1]` (Line 34). This promotion also transforms the d-dimensional polyhedron into a $(d+1)$-dimensional degenerate polyhedron.

The third and last step (Lines 36 to 37) promotes all the d-dimensional polyhedra in `vertices_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* (i.e., it is emitted to the output stream) when it is promoted to the dimension above the dimension of the space, that is, 3 for an instruction in a 2D loop nest. When the stream is finished, all remaining nonretired 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.

5.2.2 Absorption. As stated in the previous section, 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 its corner touches the anchor of any other $(d-1)$-dimensional polyhedron (Algorithm 1, Line 23). This search is performed by adding the *search vectors* `v` to the coordinates of the corner and performing a lookup in `vertices_2_to_be_absorbed[d]` to see if there is a polyhedron at this position (Line 24). Once a candidate has been found, the algorithm must check that the absorption is possible (Line 27), that is, leads to an elementary polyhedron (`has_compat_geometry`) with a correct label function (`has_compat_label`). Which search vectors are used for the lookup and how geometric compatibility 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 Figure 5. The second case, a *polyhedra extension*, occurs when the absorber is not degenerate, as seen, for example, when P1 absorbs P7. The `has_compat_geometry` function called once a candidate has been found is shown in Algorithm 2.

Polyhedra merge. In this case, the d-dimensional absorber polyhedron is degenerate on dimension d. Hence, it has no edges yet along that dimension. As a consequence, there are many

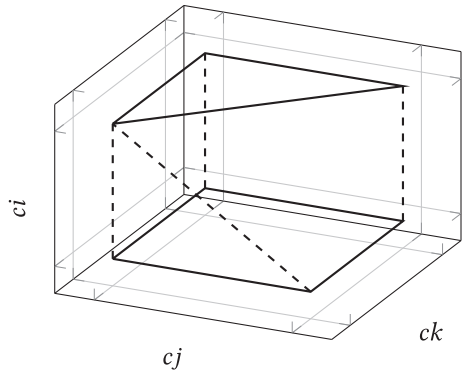

Fig. 7. Example of invalid polyhedron after absorption.

possibilities where to look for the anchor of the to-be-absorbed polyhedron. Our algorithm uses the set of all possible 3^{d-1} search vectors written as $v = (0, \ldots, 0, 1, \delta_{d-1}, \ldots, \delta_1)$, where for $i < d$, $\delta_i \in \{-1, 0, 1\}$.

ALGORITHM 2: The `has_compat_geometry` function ensures the polyhedron resulting from absorption is still an elementary polyhedron

```
1  # Geometry compatibility check
2  def has_compat_geometry(abso, to_be_abs, d):
3    # Polyhedra merge case
4    if abso.is_degenerate_on(d):
5      for k in [0, 2^(d - 1)[ :
6        diff = to_be_abs.vertices[k] - abso.vertices[k]
7        if not diff.is_a_search_vector(d):
8          return False
9      for side_face in side_faces_of_merged_polyhedron(abso, to_be_abs, d):
10       if not all_points_lie_on_same_hyperplane(d, side_face):
11         return False
12   # Polyhedra extension case
13   else:
14     for k in [0, 2^(d - 1)[ :
15       v = abso.growing_directions[k]
16       if abso.vertices[k] + v != to_be_abs.vertices[k]:
17         return False
18   return True
```

Once a candidate polyhedron has been found, the `has_compat_geometry` function call verifies that concatenating the vertices of the two polyhedra leads to a well-formed polyhedron (Algorithm 2, Lines 4 to 11). First, the function checks (Lines 4 to 8) that all the corresponding vertices of both polyhedra are connected through the search vectors used to find the anchor of the to-be-absorbed polyhedron described just above. Second, the function checks (Line 9 to Line 11) that all the side faces of the polyhedron resulting from the absorption are valid. As shown in Figure 7, even if the lower (a square) and the upper (a triangle) faces are valid elementary polyhedra, the result of the absorption may not be a valid polyhedron. For this we check if all the points of the side face lie on the same hyperplane (Line 11). To begin with, we arbitrarily designate one point of the face as the origin of the plane. We then pick $d - 1$ other points to calculate a normal vector n for the plane

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

Fig. 8. The data structure used to represent label functions.

by calculating the nullspace of the space spanned by the vectors from the origin to the other points. And finally, we verify for all remaining points p that the dot product $(origin - p) \cdot n$ equals zero.

If absorption is performed, the resulting polyhedron will no longer be degenerate in d. By construction, if well formed, the so-obtained polyhedron is necessarily an elementary d-dimensional polyhedron. Its lower face will be the original absorbing polyhedron, while its upper face will be the absorbed polyhedron.

Polyhedra extension. In this case the absorber is a nondegenerate d-dimensional polyhedron. Hence, the absorber already has edges along dimension d. When looking for candidates to absorb, there is only one search vector, the edge connecting the lexicographically smallest vertex of the lower face to that of the upper face. To check if the absorption is legal, `has_compat_geometry` simply verifies (Line 13 to Line 17) whether the vertices of the two polyhedra can be connected using the existing edges of the absorber stored in the `growing_directions` list.

5.2.3 Compatibility and Update of label functions. The absorption is performed only if both geometric and label compatibility are satisfied (Line 27). This section describes how label functions are represented, created, and combined together.

Label functions. The data structure used for label functions is shown in Figure 8. `num_dimensions` is the number of loops enclosing the static instruction or the destination instruction associated with the input stream.

Creation. Label functions are created when a new polyhedron is created from a single point (Line 13). At this time, all the coefficients of the function 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 $\top$. The lifecycle of a coefficient is then the following one:

$$\bot \rightarrow \mathbb{Z} \rightarrow \top.$$

At creation time, `coeff[0]` is given the value of the label 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 function.

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 functions where some coefficients are $\bot$. In the following, we note the label function 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 5.2.1. 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 functions. The compatibility check for this simple case is shown in Algorithm 3. First, the function `are_compat_dims_known` verifies that all coefficients for dimensions from 1 to $d-1$ are the same. If this is not the case, the two label functions are incompatible (Line 6).

Otherwise, the check may be faced with two cases corresponding to the two different absorption cases described in Section 5.2.2. 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 `are_compat_dims_known` function returns `true` (Line 11). Indeed, by setting the proper coefficient for dimension d and by updating the remaining value, it is always possible to make the two functions compatible as shown by the `update_label_dims_known` function in Algorithm 3. The new coefficient is equal to the difference of remaining values (Line 20). 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 functions 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 24).

ALGORITHM 3: Simplified version of the compatibility check and update of coefficient for the case when all dimensions below d are known. See Algorithm 4 for the general case of `has_compat_label`. The general case for `update_label` uses the same principle as the simplified case.

```
1  # Compatibility check when all dimensions below d are known
2  def has_compat_label(f_abs, f_to_be_abs, d):
3    # Verifies coefficients below d are the same
4    for q in range [1, d-1]:
5      if f_abs.coeffs[q] != f_to_be_abs.coeffs[q]:
6        return False
7    # Polyhedra merge case
8    if f_abs.coeffs[d] == ⊥:
9      return True
10   # Polyhedra extension case
11   else:
12     new_coeff_contrib = f_abs.coeffs[d] * f_to_be_abs.coeffs[d]
13     new_remain = f_to_be_abs.coeffs[0] - new_coeff_contrib
14     return new_remain == f_abs.coeffs[0]
15
16 # Update coefficient for dimension d and remaining value of f_abs.
17 # No need to update f_to_be_abs because it will be thrown away after absorption
18 def update_label(f_abs, f_to_be_abs, d):
19   # Update of coefficient
20   new_coeff = f_to_be_abs.coeffs[0] - f_abs.coeffs[0]
21   f_abs.coeffs[d] = new_coeff
22   # Update of remaining value
23   new_coeff_contrib = new_coeff * f_abs.init_point[d]
24   f_abs.coeffs[0] = f_abs.coeffs[0] - new_coeff_contrib
```

In the polyhedra extension case, the absorber polyhedron is not degenerate on dimension d. Its affine function 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`. This is done by first computing the contribution of the known coefficient of `f_abs` into `f_to_be_abs` using the initial point of `f_to_be_abs` (Line 12). 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 14).

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

ALGORITHM 4: General compatibility check for label functions

```
1  # General compatibility check
2  def has_compat_label(abso, to_be_abs, d):
3    # Loop over all pairs of label functions
4    for f_abs, f_to_be_abs in abso.label_functions, to_be_abs.label_functions:
5      # Contributions from other functions
6      abs_diff = 0
7      to_be_abs_diff = 0
8      # Loop over all coefficients
9      for q in [1, d]:
10       abs_t = f_abso.coeff_types[q]
11       to_be_abs_t = f_to_be_abs.coeff_types[q]
12       # Both coefficients known, must be the same
13       if abs_t != ⊥ and to_be_abs_t != ⊥:
14         if f_abso.coeffs[q] != f_to_be_abs.coeffs[q]:
15           return False
16       # One coefficient is not known, the other is
17       if abs_t == ⊥ and to_be_abs_t != ⊥:
18         abs_diff += f_abso.init_point[q] * f_to_be_abs.coeffs[q]
19       # One coefficient is known, the other is not
20       if abs_t != ⊥ and to_be_abs_t == ⊥:
21         to_be_abs_diff += f_abso.coeffs[q] * f_to_be_abs.init_point[q]
22     # The check
23     if (f_abso.coeff[0] - abs_diff) != (f_to_be_abs.coeff[0] - to_be_abs_diff):
24       return False
25   return True
```

Function `has_compat_label` in Algorithm 4 shows the general compatibility check between two polyhedra. The check is performed on all the matching pairs of label functions of the two polyhedra. Remember that there are several such label functions in the case of dependencies, one for each dimension of the source instruction.

The check works by comparing the coefficients of both functions 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 13). If one is known and not the other (Line 17 and Line 20), then the function increments the total contribution coming from the other function for the function having the unknown coefficient. At the end of the loop (Line 23), 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 functions 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 function of the absorber are computed by the same principle as in the simplified case from Algorithm 3.

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 15 in Algorithm 4), 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 label function of 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 functions from equalities to inequalities, as shown in Section 4.3.

The integration of this feature into Algorithm 4 is straightforward. A $\top$ coefficient is compatible with any other coefficient, and when performing absorption, any such coefficient in one of the two label functions leads to a $\top$ coefficient in the updated function.

The label widening mechanism is crucial for the label functions of instructions because $\top$ is a clear indicator that a memory access 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.

5.2.4 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. It occurs for statements surrounded by *if* conditionals in the program. 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 13), the algorithm checks if the number of intermediate polyhedra exceeds the threshold. If so, the associated input stream is marked as *give up*. Once a stream has been marked as give up, all intermediate polyhedra for that stream are discarded. The discarded polyhedra are then replaced by a hyperrectangle that starts at the origin and extends to the maximum coordinate seen in the *IVs* of any point contained in the discarded polyhedra. In other words, the geometry of the input stream is overapproximated by a single large polyhedron. 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 of the hyperrectangle are updated as necessary for every point. Lastly, all coefficients for all outputs of the label function for this input stream are set to $\top$; that is, a geometric give-up implies giving up on all dimensions of the label function.

Similar to the widening of label functions, once geometric give-up has occurred, it is no longer possible to reproduce the original input stream. However, the overapproximated geometry is guaranteed to contain all points seen in the input.

5.3 Complexity Analysis

Let us first recall the main idea of our folding algorithm. The folding process starts with polyhedra of dimensionality zero, one for each point. Then, absorption is performed dimension by dimension from innermost to outermost. As the process advances, the dimensionality of the polyhedra involved grows. It turns out that the complexity of an absorption also grows with its dimensionality. But as the dimensionality increases, the number of absorptions, that is, the number of intermediate polyhedra, also decreases. The more regularity there is, the more the number of intermediate polyhedra decreases. In other words, as formalized below, but for fully irregular programs for which

neither label widening nor geometric give-up has been enabled, one should expect an overall complexity linear in the number of input points.

More formally, in a D-dimensional space, we note the total number of input points as N and the overall number of intermediate polyhedra seen in the `absorbers[d]` list when iterating over it as $N_d, \forall 1 \leq d \leq D$ (Algorithm 1, Line 20). For a given $d \leq D$ we have:

- the number of total iterations of the `for` loop over `absorbers[d]` (Line 20) is N_d;
- for each absorber, there are at most 3^{d-1} lookups to find a polyhedron to absorb (Line 22);
- testing if absorption is possible with regards to the label criterion (`has_compat_label` Line 27) has cost $O(d)$;
- testing if absorption is possible with regard to the geometry criterion (`has_compat_geometry` Line 27) has cost $O(d \times 2^{d-1} + (2 \times (d-1)) \times (d^3 + d \times (2^{d-1} - d))) = O(d^2 \times 2^{d-1})$. Here, the first $d \times 2^{d-1}$ corresponds to checking the search vectors or growing directions. The factor $(2 \times (d-1))$ comes from the loop over the side faces of the merged polyhedron (Line 9). The term d^3 corresponds to calculating the normal vector of the hyperplane of the sideface. And the final $d \times (2^{d-1} - d)$ corresponds to checking if the remaining points of the side lie on the same hyperplane.

This leads to an overall complexity of

$$O\left(\sum_{d=1}^{D} N_d \times 3^{d-1} \times d \times 2^{d-1}\right) = O\left(\sum_{d=1}^{D} N_d \times 6^{d-1} \times d\right).$$

To illustrate the notations, let us assume a perfectly nested loop of depth D and size $N = n_D \times \cdots \times n_2 \times n_1$. Let us also consider the scenarios where the loop nest is either fully regular or geometrically regular only and label widening is enabled. In these two cases, the folding algorithm leads to a single polyhedron. We have $N = n_1 \times n_2 \times \cdots \times n_D$, $N_1 = N$, $N_2 = N/n_1$, $N_3 = N/(n_1 \times n_2)$, ..., $N_D = n_d$. The overall complexity is then

$$O\left(N + \sum_{d=2}^{D} \frac{N \times 6^{d-1} \times d}{\prod_{j=1}^{d-1} n_j}\right) = O\left(N + N \times \sum_{d=2}^{D} d \times \prod_{j=1}^{d-1} \frac{6}{n_j}\right) = O\left(N + N \times \sum_{d=2}^{D} \prod_{j=1}^{d-1} \frac{j+1}{j} \times \frac{6}{n_j}\right).$$

Observe that in practice we will almost always have $\forall 1 \leq j < D$, $n_j \geq \frac{j+1}{j} \times 6$, which leads to a complexity of $O(N)$.

Obviously, N_d being always bounded by N, we have a worst-case complexity of $O(N \times D \times 6^D)$. This worst-case scenario will occur for fully irregular input streams where every absorption fails even with label widening, that is, where absorption failures are caused by geometric incompatibility. Geometric give-up allows the algorithm to handle these input streams with a linear complexity.

6 EXPERIMENTAL RESULTS

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

Experimental setup. We use the latest revision, 3.1, of the Rodinia benchmark suite [10, 11]. All measurements and experiments were 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 were 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 5.3 speedup

Table 5. Evaluation of the Folding Algorithm

Benchmark	Dependencies												Instructions						Optim
	Input	F		F_W			F_{GG}			$F_{GG,W}$			Input	F_W			$F_{GG,W}$		
	Size	#P	$\#MP_I$	#P	%A	$\#MP_I$	#P	%A	$\#MP_I$	#P	%A	$\#MP_I$	Size	#P	%A	$\#MP_I$	#P	%A	$\#MP_I$	
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	-

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 instruction streams that form the input of the folding algorithm are actually more complicated than the ones presented in Section 4.1 and Algorithm 1 in a simplified way for clarity purposes. In reality, the label of an instruction can have multiple values to account for both the addresses accessed and the values produced. The label functions 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:

- F is the basic algorithm as described in Section 5, with label widening for instructions and without for dependencies.
- F_W is the algorithm with label widening for both instructions and dependencies.
- F_{GG} is the same as F but with geometric give-up.
- $F_{GG,W}$ is the same as F_W but with geometric give-up.

The threshold for the geometric give-up was set to allow $4d + 1$ intermediate polyhedra in each d-dimensional space—that is, enough for the affine function 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 were in an affine piece of the label function. A piece of the label function is considered affine if it has no $\top$ coefficient. This column is omitted for algorithm F since by construction it always contains 100%.

- Similarly for instructions, **%A** is the number of instruction instances that were in an affine piece of the label function. A piece of the label function of a static instruction is considered affine if it either:
  —does not perform a memory access or
  —has no $\top$ coefficient in its memory access function.
- **#MP**$_I$ 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 $F_{GG,W}$ [19]:
  —***T nD*** indicates that the back-end has found that n-dimensional tiling was possible.
  —***P*** indicates that the back-end has detected parallelism that can be exploited using threads.
  —***V*** indicates that the back-end has detected potential for vectorization.

Note that the entire feedback of the tool is immensely richer and more elaborate [19]; this column gives only a simplified summary.

Finally, note that the numbers reported in Table 5 correspond 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 [19].

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 $\mathbf{F}_{\mathrm{GG}}$ and $\mathbf{F}_{\mathrm{GG,W}}$ produce drastically smaller outputs than the other two versions. As indicated by the **%A** column, $\mathbf{F}_{\mathrm{GG,W}}$ is roughly as precise as $\mathbf{F}_{\mathrm{GG}}$ but produces an even smaller output. In fact, only the output of $\mathbf{F}_{\mathrm{GG,W}}$ is small enough for the back-end to handle.

Since Rodinia is a benchmark suite designed to exploit multicore 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.

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.

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 overapproximating the dependencies that do not fit the polyhedral model while still recovering precise information for those that do. Thanks to our overapproximation mechanisms, we are able to build a compact polyhedral representation in which we can detect the potential for several high-level loop optimizations.

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 [24, 35], 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 functions. 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 tradeoff between the accuracy and the size of the output of the folding algorithm.

REFERENCES

[1] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime pointer disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. ACM.

[2] Ran Ao, Guangming Tan, and Mingyu Chen. 2013. ParaInsight: An assistant for quantitatively analyzing multi-granularity parallel region. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC'13)*. IEEE.

[3] Cédric Bastoul. 2004. Generating loops for scanning polyhedra: Cloog users guide. *Polyhedron* 2 (2004).

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

[5] Erik Berg and Erik Hagersten. 2005. Fast data-locality profiling of native execution. In *ACM SIGMETRICS Performance Evaluation Review*. ACM.

[6] Kristof Beyls and Erik D'Hollander. 2006. Discovery of locality-improving refactorings by reuse path analysis. *High Performance Computing and Communications* 4208 (2006), 220–229.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM.

[8] G. S. Brodal and R. Jacob. 2002. Dynamic planar convex hull. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002, Proceedings.*

[9] Khansa Butt, Abdul Qadeer, Ghulam Mustafa, and Abdul Waheed. 2012. Runtime analysis of application binaries for function level parallelism potential using QEMU. In *2012 International Conference on Open Source Systems and Technologies (ICOSST'12)*. IEEE.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization, 2009 (IISWC'09)*.

[11] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. 2010. 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)*. IEEE Computer Society.

[12] Jean-François Collard, Denis Barthou, and Paul Feautrier. 1995. Fuzzy array dataflow analysis. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*. ACM.

[13] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. 2017. Optimistic loop optimization. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO'17)*. IEEE Press.

[14] Karl-Filip Faxén, Konstantin Popov, Sverker Jansson, and Lars Albertsson. 2008. Embla - data dependence profiling for parallel programming. In *Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'08)*. IEEE Computer Society.

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

[16] Paul Feautrier and Christian Lengauer. 2011. Polyhedron model. In *Encyclopedia of Parallel Computing*. Springer.

[17] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 4 (2012). https://www.worldscientific.com/doi/10.1142/S0129626412500107.

[18] Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, and Fabrice Rastello. Python implementation of the folding based analysis. Retrieved from https://gitlab.inria.fr/fgruber/python-folding.

[19] Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, and Fabrice Rastello. 2019. Data-flow/dependence profiling for structured transformations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*.

[20] Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Louis-Noël Pouchet, and Fabrice Rastello. 2019. *Building of a Polyhedral Representation from an Instrumented Execution: Making Dynamic Analyses of Non-Affine Programs Scalable*. Research Report RR-9244. Retrieved from https://hal.inria.fr/hal-01967828.

[21] Christophe Guillon. 2011. Program instrumentation with QEMU. In *Proceedings of the International QEMU User's Forum (QUF'11)*.

[22] John Hershberger and Subhash Suri. 2003. Convex hulls and related problems in data streams. In *Proceedings of the ACM/DIMACS Workshop on Management and Processing of Data Streams*.

[23] Justin Holewinski, Ragavendar Ramamurthi, Mahesh Ravishankar, Naznin Fauzia, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2012. Dynamic trace-based analysis of vectorization potential of applications. *ACM SIGPLAN Notices* 47, 6 (2012).

[24] Alain Ketterlin and Philippe Clauss. 2008. 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 (CGO'08)*. ACM.

[25] Alain Ketterlin and Philippe Clauss. 2012. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society.

[26] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. 2010. Prospector: A dynamic data-dependence profiler to help parallel programming. In *HotParâ10: Proceedings of the USENIX Workshop on Hot Topics in Parallelism*.

[27] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. 2015. DiscoPoP: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014*. Springer.

[28] Xu Liu and John Mellor-Crummey. 2011. Pinpointing data locality problems using data-centric analysis. In *2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. IEEE.

[29] G. Marin, J. Dongarra, and D. Terpstra. 2014. MIAMI: A framework for application performance diagnosis. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*.

[30] Juan Manuel Martinez Caamaño, Manuel Selva, Philippe Clauss, Artyom Baloian, and Willy Wolff. 2017. 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 cpe.4192.

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

[32] Catherine Mills Olschanowsky, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. 2010. PSnAP: Accurate synthetic address streams through memory profiles. In *Languages and Compilers for Parallel Computing*, Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li (Eds.). Springer, Berlin.

[33] Sebastian Pop, Albert Cohen, and Georges-André Silber. 2005. Induction variable analysis with delayed abstractions. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'05)*.

[34] Louis-Noël Pouchet. 2019. The PoCC polyhedral compiler collection. Retrieved from http://pocc.sourceforge.net.

[35] Gabriel Rodríguez, José M. Andión, Mahmut T. Kandemir, and Juan Touriño. 2016. Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO'16)*. ACM.

[36] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. 2003. Hybrid analysis: Static & dynamic memory reference analysis. *International Journal of Parallel Programming* 31, 4 (Aug. 2003), 251–283.

[37] Diogo N. Sampaio, Louis-Noël Pouchet, and Fabrice Rastello. 2017. Simplification and runtime resolution of data dependence constraints for loop transformations. In *Proceedings of the International Conference on Supercomputing (ICS'17)*. ACM.

[38] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. 2018. PolyJIT: Polyhedral optimization just in time. *International Journal of Parallel Programming* (Aug. 2018).

[39] Aravind Sukumaran-Rajam. 2015. *Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation*. Theses. Université de Strasbourg. Retrieved from https://hal.inria.fr/tel-01251748.

[40] Aravind Sukumaran-Rajam and Philippe Clauss. 2015. The polyhedral model of nonlinear loops. *ACM Transactions on Architecture and Code Optimization* 12, 4 (Dec. 2015), 27.

[41] Georgios Tournavitis and Björn Franke. 2010. 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 (PACT'10)*. ACM.

[42] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. GRAPHITE two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*. ACM.

[43] Robert A. Van Engelen. 2001. Efficient symbolic analysis for optimizing compilers. In *International Conference on Compiler Construction*. Springer.

[44] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. 2010. The paralax infrastructure: Automatic parallelization with a helping hand. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM.

[45] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization* 9, 4 (Jan. 2013), 23.

[46] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F. P. O'Boyle. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 1 (2014), 26.

Received February 2019; revised August 2019; accepted September 2019