Data-Flow/Dependence Profiling for Structured Transformations
Fabian Gruber, Manuel Selva, Diogo Sampaio, Christophe Guillon, Antoine Moynault, Louis-Noël Pouchet, Fabrice Rastello

To cite this version:

HAL Id: hal-02060796
https://inria.hal.science/hal-02060796
Submitted on 7 Mar 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Data-Flow/Dependence Profiling for Structured Transformations

Fabian Gruber  
Manuel Selva  
Diogo Sampaio  
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

Louis-Noël Pouchet  
Colorado State University

Christophe Guillon  
Antoine Moynault  
STMicroelectronics

Fabrice Rastello  
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

Abstract
Proﬁling feedback is an important technique used by developers for performance debugging, where it is usually used to pinpoint performance bottlenecks and also to ﬁnd optimization opportunities. Assessing the validity and potential beneﬁt of a program transformation requires accurate knowledge of the data ﬂow and dependencies, which can be uncovered by proﬁling a particular execution of the program.

In this work we develop POLY-PROF, an end-to-end infrastructure for dynamic binary analysis, which produces feedback about the potential to apply complex program rescheduling. Our tool can handle both inter- and intra-procedural aspects of the program in a uniﬁed way, thus providing interprocedural transformation feedback.

Keywords Performance Feedback, Polyhedral Model, Loop Transformations, Compiler Optimization, Binary, Instrumentation, Dynamic Dependence Graph

1 Introduction
The most effective program transformations for improving performance or energy consumption are typically based on rescheduling instructions so as to expose data locality and/or parallelism. Optimizing compilers typically attempt, via static analysis, to build a representation of the program precise enough to enable useful program transformations. A key issue faced when analyzing general-purpose languages is the ambiguities introduced by the language itself: for example the use of pointers typically restricts the ability to precisely characterize the data being accessed, in turn triggering conservative and approximate dependence analysis [27, 40, 68]. Such frameworks therefore rely on conservative assumptions, limiting the ability to reason on the legality of complex program restructuring. In this work we speciﬁcally target structured transformations, that is, a (possibly interprocedural) reordering of operations involving complex sequences of multidimensional loop transformations, including (partial) loop fusion/fission, loop tiling [11], etc.

When a region of the source program ﬁts speciﬁc syntactic and semantics restrictions, such as avoiding function calls, using only arrays as data structures, and very simple conditional statements with no indirections [16, 20], transformation frameworks such as the polyhedral model [11] showed that multidimensional loop nest rescheduling can be successfully implemented, leading to signiﬁcantly improved performance [28, 53, 68]. The input program can be massaged manually via function inlining, loop rewriting/normalization, etc. to enable such static analyses to succeed, but this is a rare scenario: In full programs and in particular those relying on external binaries visible only to the linker in compiled form, often data allocation and even the full call stack is inaccessible to static analysis.

Dynamic analysis frameworks [9, 21, 47, 63] address this limitation by reasoning on a particular execution of the program, through the analysis of its execution trace. That is, disambiguation of dependence information and accessed data is addressed by monitoring the stream of operations and accessed addresses. In general, the result is only valid for that particular execution. Such systems provide some feedback to the user, pinpointing the probable absence of dependencies along some loop, thus highlighting potential parallelism [37, 67, 70, 74] or SIMD vectorization [33].

The contributions of this work are: 1. The development of an inter-procedural intermediate representation that compacts program traces effectively while being amenable to polyhedral optimization, and all associated algorithms and implementation to make the process scalable to full programs. 2. POLY-PROF, a tool that provides optimization feedback using binary translation and instrumentation. POLY-PROF is the first framework that provides feedback on structured transformation potential for arbitrary binaries.

2 Overview of POLY-PROF
In contrast to, for instance, basic loop parallelization, structured transformations can be found even in the presence of data dependencies between loop iterations, dramatically broadening the class of loop transformations that can be applied on the program. This, however, requires detailed information about such dependencies. That is, instead of just proving the absence of dependencies one must capture and represent them in order to be able to find patterns and reason about their structure.

An overview of POLY-PROF is shown in Fig. 1. Taking a carefully generated execution trace of the program, the ﬁrst objective of POLY-PROF is to construct a useful and analyzable representation of the trace. Building such a representation constitutes our ﬁrst key set of contributions, and covers the ﬁrst three stages below. We then extended a polyhedral compiler. The goal here is not to provide automated
transformations, but instead to assist the user in figuring out where optimizations should be implemented, and importantly which ones. This corresponds to the fourth stage below that forms the second contribution of this work.

**Interprocedural control structure** To eventually reconstruct a compact and useful representation of the program trace, the first stage starts by dynamically computing the intraprocedural control flow graphs (CFGs) and the interprocedural call graph (CG) from an optimized binary. This information is then used to construct the *interprocedural loop context tree* which combines the notions of loop-nesting-trees and their equivalent for call graphs, the *recursive-component set*, both of which are input to the next stage. Their structure and construction are detailed in section 3.

**Dynamic Dependence Graph** This stage generates the actual dynamic dependence graph (DDG) [50], a trace representing both the dynamic execution of a program’s instructions as well as their data dependencies. It uses the interprocedural loop context tree to construct *dynamic interprocedural iteration vectors*, a new interprocedural representation of the program execution which unifies two abstractions: the intraprocedural schedule tree [35, 72] and the interprocedural calling-context-tree [2]. Both the DDG and the dynamic interprocedural iteration vector are described in section 4.

**Compact polyhedral DDG** The third stage compacts, or folds, the DDG into a polyhedral program representation where, essentially, sequences of individual operations in the trace are merged together into sets of points. Intuitively, when a loop is executed it generates one execution of a statement for each loop iteration and therefore one point in the trace for each execution. This stage amounts to folding those points back into a loop-equivalent representation, which itself can be the input of a polyhedral compiler. However we proceed in a general case, where folding occurs across multi-level loops but also, possibly recursive, procedure calls, leading to interprocedural folding. Details are presented in [29], and an outline of the process is presented in Sec. 5.

**DDG polyhedral feedback** The last stage of *POLY-PROF* consists of a customized state-of-the-art polyhedral toolchain, for the purpose of analyzing very large polyhedral programs. It analyzes the polyhedral DDG and suggests sequences of program transformations needed to expose multi-level parallelism and/or improved data locality, along with numerous statistics on the original and potentially transformed program including SIMDization potential. Section 6 describes this last step as well as how POLY-PROF provides human interpretable feedback to the user.

Section 7 illustrates this feedback through a few case studies. Section 8 thoroughly evaluates POLY-PROF on the Rodinia benchmarking suite. Related work is discussed in Section 9 before concluding.

### 3 Interprocedural Control Structure

As most of the execution time of a program is usually spent in loops, POLY-PROF has been designed to find loop-based transformations. In practice, however, as illustrated in Ex. 1 of Fig. 3, interesting loop nests are often spread across multiple functions over long call chains, obfuscating them to traditional static analysis. POLY-PROF is designed to be able to represent both classical loops as well as function calls and returns in a unified way with polyhedra [23, 75].

POLY-PROF has been designed to handle several scenarios. 1. Nested loops containing function calls with side effects: here calls can be viewed as atomic instructions, but profiling the storage locations they access or modify and compute the dependencies between each call and the rest of the loop body is required. 2. Loops containing function calls that themselves contain loops, as in the Ex. 1 of Fig. 3: here the objective is to view the whole interprocedural region as a multidimensional loop nest. 3. Recursive functions, as in Ex. 2 of Fig. 3: here the primary objective is to avoid the depth of the generated representation to grow proportionally with the depth of the call stack; A secondary objective is to detect any underlying regular loop structure amenable to polyhedral compilation (after recursion-to-iteration conversion).

This section explains how to go from the execution of a program to a representation of its interprocedural control structure.
The first step of POLY-PROF is to extract the inter- and intraprocedural static control structure from a program in compiled binary form. This static control structure of the program will then be used in a later stage of POLY-PROF to transform the stream of raw events (jump/call/return) gathered during execution into loop events (entry/iterate/exit).

POLY-PROF performs this through instrumentation of jump, call, return and a few other instructions to dynamically trace any transfer of control during execution. It thus "dynamically" extracts the control-flow-graph (CFG) of each function and then proceeds to statically build the loop-nesting-tree. It also dynamically extracts the call-graph (CG) of the whole program and builds the counterpart of loop-nesting-trees for call-graphs, the recursive-component-set. An advantage of our approach is that only the part of a program that is actually executed will be analyzed. For large programs with only a small performance-critical part this can substantially reduce the amount of work for the analyzer.

3.1 Control-flow-graph and loop-nesting-tree

For the profiled CFG, the loop detection algorithm used by POLY-PROF is the one described by Havlak [31]. As formalized by Ramalingam in [58], a loop-nesting-forest can be recursively defined (the actual algorithm is almost linear) as follows: 1. Each strongly connected component (SCC) of the CFG containing at least one cycle constitutes the region of an outermost loop; 2. For each loop, one of its entry nodes is designated the header node of the loop; 3. All edges within the loop that target its header node are called back-edges; 4. "Removing" all back-edges from the CFG allows one to recursively define the next level of sub-loops.

An example CFG and its corresponding loop-nesting-tree is given in Figures 2a & 2b. Here, the CFG can be partitioned into one SCC (giving rise to loop L1 with B as its header) plus two separated basic-blocks A and E. The region of L1, once its back-edge (D, B) has been removed, contains one sub-SCC formed by the loop L2 and the basic-block B. Among the two entry nodes C and D of L2 only one, C, is selected to be its header. As depicted in Alg. 1, those notions (region, header, back-edge) are important as they allow associating loop events (E: entry, l: iterate, and X: exit) with the stream of raw control events (jump, call, return) that are produced by our instrumented execution. Here, generation of loop events is driven by the control event "jump" that is emitted by our instrumentation. Whenever we detect the start of a new iteration of a given loop (Line 8), all live sub-loops are considered exited, that is, we emit an "exit" event (Line 4). This is especially important for recursive loops as further explained in Alg. 2.

3.2 Call-graph and recursive-component-set

To uniquely identify each dynamic execution of an instruction, also called a dynamic instruction, POLY-PROF uses interprocedural iteration vectors. This is described in more detail in Section 4. Note, that the modeling of programs containing recursive function calls with calling-context-paths is memory inefficient since the length of the paths is proportional to the depth of the recursion. While recursive function calls are found to be very uncommon in performance critical code, we do need to handle them to ensure robustness. POLY-PROF handles recursiveness in an elegant way that we believe to be useful beyond the restricted scope of this paper, for example to detect properties of tree-recursive calls. The main data structure for treating recursive control flow is the recursive-component-set which is for the call-graph what the loop-nesting-tree is for the control-flow-graph.

Before providing more details, we go through the illustrative Ex. 2 of Fig. 3. Here, the edge from B3 to B0 (in orange) is not a CFG-edge but a recursive call to B from call site B3. The recursive-component-set computed from the CG contains one SCC with a cycle made up of a single function B. This example raises several questions: 1. Should C0 be part of the associated recursive loop? It actually depends on the context: It should, when called from B1, while it should not when called from D0. 2. What about B5? One should observe that B5 will be executed as many times as the number of recursive calls to B: In other words, B5 should be part of a loop.

Algorithm 1 CFG-loop events generated from a jump event.

Input:
- event: Branch event.
- I(L, H): entry into loop L, H is the header.
- X(L, H): exit of loop L, H is the header.
- X(L, B): exit of loop L, jumping to basic-block B
- N(B): local jump to basic-block B.

Emit events:
- I(L, H): entry into loop L, H is the header.
- X(L, B): exit of loop L, jumping to basic-block B
- N(B): local jump to basic-block B.

1 if event.type is local - jump:
2 B:=event.dstBB
3 while L:=inLoops.peek() and L.isCFG and B not in L:
4 L.visiting := false; inLoops.pop(); emit X(L, B)
5 if B is loop - header:
6 H:=B; L:=loop(H)
7 if L.visiting := false:
8 L.visiting := true; inLoops.push(L); emit E(L, H)
9 else: emit I(L, H)
10 emit N(B)
11 else:
12 ... # Alg. 2
To conclude, while CG-cycles clearly represent potential dynamic loop structures, a CG-edge does not have the same semantic as a CFG-edge. In particular, a call will never exit a recursive loop. For POLY-PROF, the recursive loop "L_t" of Ex. 2 is a dynamic notion defined as follows: 1. Entering L_t is characterized by a call to B (step 1). 2. L_t is exited when this (first) call gets unstacked, that is, when the number of returns reaches the number of calls (step 22). 3. Everything executed in between is part of the loop and iteration, and the corresponding increment of induction variables takes place whenever a call/return to/from B occurs (steps 10,15,20, and 21). As one can see, once the recursive-component-set has been computed from the CG, the only relevant information dynamically used to devise recursive-loop events corresponds to the header functions, here B.

As already stated, the recursive-component-set is for the CG what the loop-nesting-tree is for the CFG. In a similar way it can be recursively defined as follows:

1. Find all the top-level SCCs with at least one cycle in the CG. Each gives rise to a distinct recursive-component.
2. For each component, determine all its entry nodes.
3. Repeat the following phases until no more cycles exist:
   a. For a given SCC, choose an entry node and add it to the headers-set of the recursive-component (top-level SCC) it belongs to.
   b. Remove all edges inside this SCC that point to this node.

Havlak’s loop-nesting-forest construction algorithm can easily be adapted to build the recursive-component-set in almost linear time. The end result of the algorithm is a possibly empty set of recursive-components where each recursive-component has a non-empty set of headers plus a non-empty set of entries associated with it. Alg. 2 uses this data structure to associate loop events (entry, iterate, and exit) to control events (call and return). Here: 1. entering a recursive loop is characterized by a call to a component’s entry function (Line 4); 2. a new iteration is started whenever a call/return to/from one of the components’ header occurs (Line 6); 3. an exit occurs when all the iterating calls to the headers have been unstacked, that is, when the number of returns equals the number of calls, and we are returning from the original function that entered the loop (Line 18). Tracking the state of the call stack is done with the following two data structures: L.stackcount represents for a recursive-component L a counter of the number of calls-to a header minus the number of returns from it; L.entry represents the function through which L was entered.

4 GG: Dynamic dependence graph

The objective of the second stage of POLY-PROF ("Instrumentation II") is to profile the dynamic dependence graph of a given execution, that is, to build a graph that has one vertex for every dynamic instruction and one edge for every data dependence. Because we want to enable feedback with structured loop transformations, we need to map this graph to a “geometric” space that reflects the structural properties of the program. To this end, we tag each dynamic instruction with its iteration vector (IV). The IVs uniquely identify each dynamic instruction and naturally span a geometric space.

A data dependency is then simply represented as the pair of the IVs of the producer and the consumer.

To handle interprocedural programs we also need a notion of calling context that is scalable in the presence of recursive calls. Our dynamic interprocedural iteration vector (dynamic IV) described in this section addresses those objectives by unifying two notions: 1. Kelly’s mapping which describes interprocedural IVs and is used by the polyhedral framework [35]; 2. calling-context-paths used by profiling feedback tools. We first briefly recall those two notions.

Kelly’s mapping For a given function, Kelly’s mapping can be explained using a form of schedule tree [72] as shown in Fig. 4. Here a schedule tree is nothing else than a decorated loop-nesting-forest. The first decoration consists of associating a “static” index to each loop and basic-block: Recall the recursive characterization of loops given by Ramalingam in the previous section. For a given loop region (e.g. L_i) in the schedule-tree of the fused version in Fig. 4), its sub-regions (once back-edges have been removed – here statements S and T) form a directed-acyclic-graph (reduced DAG represented in dashed in Fig. 4) that can be numbered using a topological order. This numbering is used to index the corresponding nodes at every level of the loop-nesting-tree (e.g. S(0) and T(1) for the fused schedule or L_i(0) and L_i(1) for the fissioned one). The second decoration consists of associating a canonical induction variable, that is, an induction variable that starts at value 0 and increments by 1, to each loop-vertex. As an example the loop-vertex associated with L_i is labeled
with \( L_j \) as well as the static index 0 followed by its canonical variables \( j \), resulting in \( L_j(0), (j) \). For any given statement, an IV with Kelly’s mapping is nothing else than the vector given by the path from the root to its corresponding leaf, called its \textit{iteration vector}. Fig. 4 shows this mapping both in its textual form using region names and numerical form using indices. As one can see, an interesting property of the numerical form of this mapping is that the scheduling of the original code is given by the lexicographical order of the so obtained iteration vectors (unique per dynamic instance).

**Calling-context-tree** Differently from the schedule tree, the calling-context-tree [2] (CCT) is only enumerative (it does not contain any loop indices) and reflects a dynamic behavior. In other words, it encodes the dynamic-call-tree in a “compact” way. The calling-context-tree of the example in Fig. 3f is reported in Fig. 3h. This figure is slightly different from the “original” CCT, but corresponds to the current practice: to differentiate two calls to a common function from different basic-blocks, callees are labeled with call sites (in purple and under parenthesis in Fig. 3h). In this example, a calling-context-path (for example \( M_1/B_3 \ldots B_j/B_i/C \)) can be as long as the number of recursive calls to \( B \), but the calling context is fully encoded making it possible to differentiate the different contexts within which basic-block \( C \) is executed. Obviously, looking at this example, one wants to fold all the repeated calls from \( B_i \) to \( C \).

**Dynamic IV** The dynamic IV is basically a combination of Kelly’s mapping and the CCT. Similarly to Kelly’s mapping, the dynamic IV alternates between context-ids and canonical induction variables. Differently from Kelly’s mapping, but more like the CCT, each context-id in a dynamic instance is a, possibly empty, call stack (e.g. \( A \)) or statement-ids (e.g. \( B \)) and each context-id is decorated with \( L_j \) as well as the static index 0 followed by its canonical variables \( j \), resulting in \( L_j(0), (j) \). For any given statement, an IV with Kelly’s mapping is nothing else than the vector given by the path from the root to its corresponding leaf, called its \textit{iteration vector}. Fig. 4 shows this mapping both in its textual form using region names and numerical form using indices. As one can see, an interesting property of the numerical form of this mapping is that the scheduling of the original code is given by the lexicographical order of the so obtained iteration vectors (unique per dynamic instance).

**Calling-context-tree** Differently from the schedule tree, the calling-context-tree [2] (CCT) is only enumerative (it does not contain any loop indices) and reflects a dynamic behavior. In other words, it encodes the dynamic-call-tree in a “compact” way. The calling-context-tree of the example in Fig. 3f is reported in Fig. 3h. This figure is slightly different from the “original” CCT, but corresponds to the current practice: to differentiate two calls to a common function from different basic-blocks, callees are labeled with call sites (in purple and under parenthesis in Fig. 3h). In this example, a calling-context-path (for example \( M_1/B_3 \ldots B_j/B_i/C \)) can be as long as the number of recursive calls to \( B \), but the calling context is fully encoded making it possible to differentiate the different contexts within which basic-block \( C \) is executed. Obviously, looking at this example, one wants to fold all the repeated calls from \( B_i \) to \( C \).

**Dynamic IV** The dynamic IV is basically a combination of Kelly’s mapping and the CCT. Similarly to Kelly’s mapping, the dynamic IV alternates between context-ids and canonical induction variables. Differently from Kelly’s mapping, but more like the CCT, each context-id in a dynamic IV is a, possibly empty, stack of calling contexts and the identifier for a basic block.

Examples of IVs are shown in Fig. 3a: Function \( A \) contains a loop \( L_1 \) that contains a call to function \( B \), itself containing a loop \( L_2 \). In this interprocedural example one clearly wants to see that basic-block \( B_1 \) in loop \( L_2 \) belongs to a two-dimensional nested loop. This is reflected by our dynamic IV (see steps 5,7 of Fig. 3d) which will be \((M_0/L_1, i_1, A_1/L_2, i_2, B_1)\) where \( i_1 \) (resp. \( i_2 \)) are the canonical induction variables of loop \( L_1 \) (resp. \( L_2 \)). Here, our context-ids are loop-ids (e.g. \( L_2 \)) or statement-ids (e.g. \( B_0 \)) and each context-id is decorated with a, possibly empty, call stack (e.g. \( A_1 \)).
1 for (i=0; i<n; i++)
2 for (j=0; j<i; j++)
3 ( S; T )
4 for (i'=0; i'<n; i'++)
5 for (j'=0; j'<i'; j'++)

(a) Nested loop before and after fission

(b) Corresponding schedule trees (reduced DAG in dashed)

(c) Corresponding Kelly’s mapping / iteration vector (IV)

Figure 4. Schedule tree (pink: static index; purple: induction vars) and Kelly’s mapping

<table>
<thead>
<tr>
<th>interprocedural</th>
<th>loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCT / calling-context-path</td>
<td>X</td>
</tr>
<tr>
<td>dy. sched. tree / dynamic IV</td>
<td>✓</td>
</tr>
</tbody>
</table>

Algorithm 3: Updating of dynamic IV (diiv)

Input:
- event: Branch event
- diiv: Current dynamic IV.

Output:
- Updated dynamic IV

1 if event == C(L, B) or E_c(L, B):
2 diiv.addDimension(0, B)
3 if event == E(L, B) or E_L(B, L):
4 diiv.addDimension(1, B)
5 if event == X(L, B):
6 diiv.removeDimension()
7 diiv.addDimension(0, B)
8 diiv.addDimension(1, B)
9 if event == I(L, B) or I_c(L, B):
10 diiv.addDimension(1, B)
11 if event == R(B):
12 diiv.addDimension(0, B)
13 diiv.addDimension(1, B)
14 diiv.addDimension(0, B)

5 Compact Polyhedral DDG

In the DDG, every dynamic instruction is represented by one vertex and every data dependence by one edge. A program running on a modern CPU for just a few seconds can create billions of nodes in the DDG. Our folding algorithm compresses those large graphs into a form that is amenable to polyhedral analysis performed by our POLY-PROFS back-end. The details are presented in a separate paper [29]. Only the
Some of the streams compressed by compilers as scalar evolution expressions (SCEVs) [54, 69], as they can be expressed as functions of the canonical induction variables. Detecting SCEVs is important for two reasons: First of all, the chains of dependencies associated to their computation shall be ignored, as otherwise it greatly and unnecessarily constrains possible code transformations; Second, it allows to detect an important class of memory access patterns, the strided accesses, that are used to evaluate the profitability of some loop transformations.

If we detect during folding of an instruction that all the Label values of a polyhedron can be expressed using a SCEV, that is, if the folding succeeds in constructing an affine function to express them, the corresponding points in the folded polyhedra of IIVs as well as the associated dependencies are removed from the DDG. This happens for example for instructions 15 and 18. For instruction 15, the folding algorithm finds a SCEV with value \(a(cj,ck) = 0cj + 1ck + 1\).

### Over-approximations

Even in programs where the hot region is affine such as in PolyBench [56], profiling the entire benchmark reveals a large amount of non-regular parts. Keeping an accurate representation of non-regular parts causes scalability issues both in the profiling part and in the polyhedral back-end of POLY-PROF. Our idea to handle such non-regular parts in a scalable fashion is through over-approximation. This over-approximation, detailed in [29], concerns labels for non-affine dependencies or non-affine SCEVs and polyhedra for dependencies and instructions with iteration domains with holes.

### 6 DDG Polyhedral Feedback

An essential motivation for folding DDGs into polyhedral structures is to enable the use of advanced polyhedral compilation systems, which are capable of finding a schedule that maximizes parallelism, finds tiling opportunities, etc. [11].

#### Polyhedral compilation of folded-DDGs

Typically, a polyhedral compiler is applied to small numerical kernels made of a handful of statements [11, 24, 48, 57]. Polyhedral schedulers suffer scalability challenge for larger programs [48]: their complexity typically grows exponentially with the number of statements in the input program. Our DDG folding and over-approximation techniques allow going from programs with thousands of statements (vastly exceeding the typical program scale these schedulers can handle) to only a few hundreds.

Numerous customizations for scalability of the polyhedral compiler have been implemented, ranging from constraining the space of legal schedules to accelerate the scheduling process to approximation of the code generation process to...
quickly output a decorated simplified AST describing the program structure after transformation. For example, the presence of large integer constants causes combinatorial blow up in the ILP solver used to solve the scheduling problem [52]. We implemented a parameterization of iteration domains, to replace those constants by a parameter (an unknown, but constant integer value). That is, a domain \( \{ i \mid 0 \leq i < 1024 \} \) is replaced by \( \{ i \mid 0 \leq i < n \land n \geq 1024 \} \) prior to scheduling. We control the number of parameters introduced by reusing the same parameter for a range of values, that is, if the value \( x \in [1024 - s, 1024 + s] \) with \( s \in \mathbb{Z} \) (we typically set \( s = 20 \)), then we replace \( x \) by \( n + (x - 1024) \).

The reader may refer to the available implementation in PoCC [55] for further details about the simplifications implemented, computation of profitability metrics is implemented in the PolyFeat module.

**Final output** The main visual support used for reporting aggregated feedback is the dynamic schedule-tree described in Sec. 4, along with a simplified AST that shows the code structure after the application of the suggested structured transformation. This AST embeds various metrics on instruction count, loop properties (parallelism and tilability) and the list of statements surrounded by every loop. This lets the user visualize the potential effort in manually writing the code structure corresponding to the application of the suggested transformation.

To better visualize the various program regions, metrics can weight each tree-node and be rendered with flame-graphs [26]. An example discussed in Sec. 7 is reported in Fig. 7. Here, the “surface” of a region in the flame graph is proportional to its estimated computation weight. Such flame graphs are SVG files, which can be clicked to obtain detailed information on each region/box. In the example of Fig. 7, loop/call nodes are marked with the label loop/call, respectively. Colors and gradients are used, that is, non-interesting regions can be grayed out. In Fig. 7, grayed regions are non-affine and blacklisted (initialization and extensive calls to libc) ones. Node width is used to highlight hotness of regions. In our example, functions adjustweight and layerforward take substantially more space than other regions.

Numerous additional extensions have been developed, to provide useful feedback to the user: various metrics about the size and structure of polyhedral regions in the DDG, before (original program) and after (transformed program) applying an optimizing schedule, the possible code structure including the precise fusion/distribution scheme, a simplified sequence of core loop transformations that should be applied to achieve the target transformation, etc. Given the extensive textual length of the feedback we provide, an example is shown only in the supplementary document.

### 7 Case Studies

This section aims to present case studies that illustrate both the type of feedback provided by **POLY-PROF** as well as the different transformations it can suggest. Current QEMU[7], on which **POLY-PROF** is based, cannot handle newer AVX instructions. Feedback was obtained from binaries compiled with GCC 8.1.1 and the flags `-g` `-O2` `-msse3`. Speedup measurements are using Intel icc and ifort compilers (version 18.0.3, flags `-Ofast` `-march=native` `-mtune=native` on a machine with a Xeon Ivy Bridge CPU with two 6 core CPUs (24 hyperthreads), each running at 2.1Ghz (GFlop/s are averaged over 50 runs).

**Case study I** This case study illustrates a simple (structured) feedback: pinpointing that dependencies live within the first quartant [4] allows exposing data locality and fine-grain parallelism through a loop interchange. For this study, we selected backprop, a supervised learning method used to train artificial neural networks extracted from the Rodinia benchmark suite [15]. In addition to what was outlined in the previous case study, **POLY-PROF** exploits its ability to associate an iteration vector with each memory access, and to match pointer values with scalar evolution. Since the scalar evolution expressions for a memory access describe the addresses it reads/writes, they can be used to detect strided accesses along any dimension, which in turn can be used to detect vectorization potential.

The flame-graph provided by **POLY-PROF** is reported in Fig. 7 where the regions of interest are: 1. the first call (of two) to bpnn_layerforward with a constant value of \( n = 16 \); 2. the last call (of two) of bpnn_adjust_weights with a constant value of \( n = 16 \). Both functions contain a 2D-nested-loop, called \( L_{\text{layer}} \) and \( L_{\text{adjust}} \) respectively, for which **POLY-PROF** suggests an interchange and SIMDization. For example, see \( L_{\text{layer}} \) on Tab. 3 for which the two loop dimensions are reported to be on lines 253 (outer) and 254 (inner) of file backprop.c: The loop nest is fully permutable, that is, interchange is possible; only the outermost loop is parallel; and there are more stride 0/1 accesses along the outermost dimension (100%) than along the innermost (67%).

To enable the suggested transformations, one needs to specialize the two interesting function calls and array expand the scalar sum. The other calls to bpnn_layerforward and bpnn_adjust_weights are left unchanged, since they (a) take up much less of the overall program runtime (b) have different arguments and do not profit from the transformation. Applying the transformation improved, in our case, bpnn_layerforward from 0.5 GFlop/s to 2.8 GFlop/s and bpnn_adjust_weights from 0.3 GFlop/s to 5.1 GFlop/s.

**Case study II** This case study illustrates an advanced feedback: providing exact dependence vector “directions” allows exposing data locality and coarse-grain parallelism through loop skewing and tiling. For this study, we selected GemsFDTD, a finite difference time domain method from the SPEC CPU2006
The dynamic analysis is based on the QEMU-plugin instrumentation interface [30]. We have extended the interface itself to support multiple interacting C/C++ plugins. Plugins primarily work at the level of the generic QEMU compiler instructions, making them CPU architecture agnostic. The choice of using QEMU for instrumenting the code is orthogonal to our contribution. There are some other candidates (e.g. Valgrind [51] or pin [45]) for implementing a dynamic binary analysis tool, each having their own advantages and disadvantages. Since we use a shadow memory to track data dependencies, dynamic analysis obviously does not come for free. As an example, the total CPU time (summing for all cores) on our server required by the first three stages of POLY-PROF to analyze the full Rodinia benchmark suite is 3h 6’ (the full execution including libc was instrumented).

We have built our polyhedral feedback pass described in Sec. 6 in the PoCC compiler [55] that implements the PhiTo scheduler [11], along with a new polyhedral DDG analysis and optimization in PoCC/PolyFeat [55]. Extensions for scalability have been implemented in the schedulers and code generator, and in PolyFeat.

**Experiment II** To show problems static approaches encounter with the Rodinia suite we also ran the static LLVM-based [40] polyhedral compiler Polly [28] over the entire suite. We used Polly version 7.0.1 and the flags -O3 -ffast-math -polly-process-unprofitable. Kernels that span multiple functions were inlined to allow Polly to see the same code region as POLY-PROF. Calls to functions from libc or the OpenMP runtime were not inlined. Where such calls are present this usually results in Polly being unable to analyze the kernel, though it can handle calls to simple functions such as exp or sqrt. Polly was unable to build a polyhedral model of the whole region of interest for any of the 19 benchmarks. It was able to model some smaller subregions, 1D or 2D loop nests, in most benchmarks, but in nearly all cases its own profitability metric decided not to optimize them. Two notable exceptions are the heartwall and lud benchmarks. In heartwall Polly was able to model a sequence of nine 2D loop nests which accounts for roughly two thirds of the code in the body of the kernel. For lud it managed to model the whole inner 3D loop nest of the kernel, but not the outermost loop. In all benchmarks the inability to model the outermost loops blocks Polly from exploiting the thread level parallelism inherent in the Rodinia benchmark suite.

**Summary statistics.** Table 5 presents summary statistics about the Rodinia 3.1 (CPU only) benchmarking suite, that we computed/aggregated by processing the feedback from POLY-PROF and Polly on each benchmark.

Table 3. backprop case study. Reported lines (e.g. 52, 253, ...) are from debug information. Suggested interchange is represented using a permutation of code lines. Statistics/properties per loop dimension as follows: (outer, inner).

<table>
<thead>
<tr>
<th>Fat regions</th>
<th>% ops</th>
<th>interchange+SIMD</th>
<th>parallel</th>
<th>permutable</th>
<th>% stride 0/1</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>backprop_kernel.c:52 (LLayer)</td>
<td>14%</td>
<td>(yes, no)</td>
<td>(100%, 50%)</td>
<td>(yes, yes)</td>
<td>(100%, 50%)</td>
<td>5.3 x</td>
</tr>
<tr>
<td>backprop_kernel.c:57 (Ladjust)</td>
<td>46%</td>
<td>(yes, no)</td>
<td>(100%, 50%)</td>
<td>(yes, yes)</td>
<td>(100%, 50%)</td>
<td>7.8 x</td>
</tr>
</tbody>
</table>

Table 4. GemsFDTD case study. Reported lines are shifted debug info.

8 Experiments

The goal of this section is to demonstrate that POLY-PROF can be systematically applied on a full benchmark suite, and find potential for optimization. Note that the output of POLY-PROF for each benchmark is extensive: flame graph, statistics on each sub-region, potential structured feedback(s), simplified annotated AST after the application of the transformation, complete AST, etc. Consequently, we only illustrate here the application of POLY-PROF on Rodinia by using aggregate metrics. These metrics are not meant to be used as is by the end user: instead, the user is expected to work on one benchmark at a time, and navigate the feedback we provide.

**Experiment I** In addition to the case studies, we evaluate below our tool-chain using the latest revision (3.1) of the Rodinia benchmark suite [14, 15]. As POLY-PROF does not support multithreaded applications yet, each benchmark is manually modified to run in a single thread, and compiled using the same compiler and flags as the case studies (GCC 8.1.1, -g -O2 -msse3).
for heatwall, hotspot, and l1d is the consequence of not supporting
lattices at folding time: These programs contain
hand linearized nested loops whose bounds use modulo
expressions and so are not recognized as fully affine. Note
that, even when parts of a benchmark are not affine, we can
still find affine over-approximations for those regions, and
potentially find transformations for the program as a whole.

Based on the statistics provided by poly-prof, the biggest
region for which the optimizer suggests a transformation
has been selected by hand. The code reference is reported
in the column Region. We considered a region to be inter-
procedural (column interproc.) if inlining was required to
perform the transformation or if it contained a call to libc or
the OpenMP runtime. Column %ops reports the percentage of
dynamic operations of the program executed inside the
region, while %Mops and %FPops reports the percentage of
memory (resp. floating point) operations of the region itself.
Note that the sum of %Mops and %FPops can be greater
than 100% since on x86 a single instruction can both load and
store to/from memory and perform an operation.

The column Reasons why Poly failed lists the reasons why
LLVM Polly was unable to model the whole region as an
affine program. They are coded as:
R. unhandled user call;
C. complex CFG (break/return);
B. non-affine loop bound or non-affine conditional statements;
F. non-affine access function (includes pointer indirection);
A. unhandled possible pointer aliasing;
P. base pointer not loop invariant.

The next group of metrics shows what can be achieved
via semantics-preserving structured transformations. skew
displays whether skewing was used in the proposed transfor-
mation, and we tend to avoid skewing unless it really provides
improvements in parallelism and tilability. %ops gives the
percentage of dynamic operations that can be parallelized us-
using OpenMP parallel pragmas. If a non-inner loop dimension
is detected as parallel, then all its operations are considered
to be parallelizable. As a loop has at least two iterations,
at least two parallel blocks can be exposed when a loop is rep-
orted parallel. Similarly, %simdops reports the percentage
of operations that occur in parallel innermost loops.

The %reuse/%Preuse metrics report space locality that is
available in the program: %reuse is the percentage of load/store-
operations that can be made stride-0 or stride-1 via a sequence of loop permutations.
We report the maximal loop depth of the region in the
source code (ld-src) and in the binary code (ld-bin). This
shows whether the compiler performed any transformation
that modifies the loop depth (e.g., full loop unrolling for cfd).
Next the maximal tiling depth (TileD) is reported, along with
%Tilops, the percentage of operations that can be tiled.

As soon as a region can be tiled, coarse-grain (wavefront)
parallelism is possible, and data reuse could be improved.
 poly-prof does not currently provide feedback on temporal
locality potential, but as illustrated in the backprop case
study %reuse allows to evaluate spatial locality improve-
ments through tiling/interchange.

Finally, metrics C/Comp./fusion outline the complexity of
the loop fusion/distribution structure that originates from the
structured transformation proposed, and is an indication
of the difficulty to manually implement a transformed code.
Any outermost loop with more than 5% of the total region
operation counts as one “component”. For example, if the
region is made of two consecutive loop nests executing each
half of the operations, then 2 components will be reported. C
reports the number of components in the binary code: Comp.
the number of components after applying the proposed struc-
tured transformation, using the fusion heuristic reported in
fusion (M for maximal loop fusion, and S for smartfuse, a
somewhat balanced fusion/distribution strategy).

Note that streamcluster exhausted memory at the sched-
uling stage, and therefore no result is displayed.

9 Related Work

This section describes previous work related to profiling,
performance debugging, and trace compression.

Performance feedback tools Using profiling feedback is
a widely used technique both for performance debug-
ging and for guiding optimization [64] (FDO). Tools such as
Perf [18] or Intel VTune [59] gather hardware counter values
and, using sampling, provide statistics (such as instruction
throughput or cache miss rate) at the granularity of machine
instruction. Various interfaces such as HPCToolKit [1] allow
navigating through the gathered statistics thanks to a best-effort mapping from binary to source code. Compiler based profiling tools, such as gprof [25], make it possible to provide complementary statistics (such as branch probabilities). While these allow pin-pointing important parts of the code where the programmer or compiler should focus his attention on, the metrics they report are very low level and consequently they leave the role of finding optimizing code transformations to the programmer or compiler heuristics [65]. Recent work such as in MAQAO [13, 19], AutoSCOPE [65] or MIAMI [46] combines these metrics with static analysis of binary/source code. Intel IACA [17], a purely static analysis tool, uses a precise machine model, which the vendor does not publish, to pin-point resource bottlenecks and predict performance of snippets of binary code.

Dynamic data-flow analyses have been presented to provide useful performance feedback. The detection of parallelism (such as vectorization) along canonical directions has particularly been investigated [3, 12, 21, 37, 38, 41, 67, 70, 74], as it requires only relatively localized information. Another use case is the evaluation of effective reuse [8, 10, 44, 46] with the objective of pinpointing data-locality problems.

The APOLLO [34, 47] framework uses an interesting technique that corresponds to detecting dynamic regularities (affine expressions) of memory accesses in nested loops, and use this information to perform speculative loop transformation, which differs in scope and challenges from the profiling feedback tool developed in this paper.

**Polyhedral compilation** Integer linear algebra is a natural formalism for representing the computation space of a loop nest. The polyhedral framework [23] leverages, among others, operators on polyhedrons (e.g. union, intersection, projection), enumeration (for code generation [6]), and parametric integer linear programming [22] (for dependence analysis [16]). 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 (e.g. Graphite [53, 68] for GCC [27] and Polly [28] for LLVM [40]). The set of loop transformations (known as affine transformations) that the polyhedral model can perform is wide and covers most of the important ones for exposing locality and parallelism to improve performance [11]. Tools such as PoCC [55] provide a convenient interface to the numerous existing state-of-the-art scheduling heuristics and polyhedral libraries such as ISL [71].

**Dynamic dependence graph** Shadow memory [76] records a piece of information for each storage location used in a program. For dependency tracking this is usually the last statement or dynamic instruction that modified that location. But shadow memories are also a core component of memory error debugging tools [51, 62].

Like PLY-PROF, Redux [50] builds a data-flow graph from binary level programs. It has, however, no notion of loops or calling contexts and does not try to compress the produced graph and is consequently only able to handle very small programs. A few techniques exist to address the high overhead of monitoring data dependencies. The most common is the use of static analysis so as to remove redundant instrumentation [42], or to avoid monitoring must-dependencies [37]. Another approach that leads to over-approximation, consists of reducing the size of the shadow memory through the use of signature based addressing [43]. The most sophisticated technique is the one developed in SDT [39] that amounts to detecting strided memory accesses to compress the shadowing. A last technique that could be used for our purposes is the parallelization of shadowing as done in [42, 49].

### Calling context tree
Using calling context trees to disambiguate instructions in different calling contexts is an idea from Ammons, Ball and Larus [2]. But in the presence of recursive functions the size of their CCTs grows proportionally with the depth of the recursion, leading to an unreasonably high memory overhead. Loop-call context trees [60] simply encode the calling context of intraprocedural loops and suffer from the same size problems in recursive programs.

### 10 Conclusion and Future Work
In this paper we introduced PLY-PROF, a profiling-based polyhedral optimization feedback tool. PLY-PROF tackles the problem of dynamic data dependence profiling and polyhedral optimization from binary programs, while addressing the associated scalability challenges of handling traces with billions of operations, as we demonstrated. Our tool handles non-regular programs with recursive function calls. Numerous technical contributions as presented in this paper were required to enable structured transformation feedback on binary programs, a significant step in complexity compared to prior approaches typically limited to unstructured transformations. We implemented numerous feedback reporting schemes for the user in PLY-PROF: flame graphs, statistics on each sub-region, proposed potential structured transformation, simplified annotated AST after the application of the transformation, complete AST, etc.

While this represents an important step towards being able to provide polyhedral feedback on full-scale applications there are issues that still need to be addressed. This includes the development of under-approximation schemes in the DDG, as well as overall scalability enhancements, like using approximate (non-optimal) polyhedral scheduling strategies. Lastly, the process of mapping our results back from the machine code back to the source code is in itself a research topic, and currently we do not provide much more than what objdump does. Our ongoing efforts in this direction leverage polyhedral program equivalence techniques [5, 61, 73].

### Acknowledgements
This work was supported in part by the U.S. National Science Foundation awards CCF-1645514, CCF-1731612 and CCF-1750399, and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d’avenir.