Optimistic Loop Optimization

Johannes Doerfert  
Saarland University  
Saarland Informatics Campus  
Saarbrücken, Germany  
doerfert@cs.uni-saarland.de

Tobias Grosser  
Department of Computer Science  
ETH Zurich, Switzerland  
tobias.grosser@inf.ethz.ch

Sebastian Hack  
Saarland University  
Saarland Informatics Campus  
Saarbrücken, Germany  
hack@cs.uni-saarland.de

Abstract

Compilers use static analyses to justify program optimizations. As every optimization must preserve the semantics of the original program, static analysis typically fall-back to conservative approximations. Consequently, the set of states for which the optimization is invalid is overapproximated and potential optimization opportunities are missed. Instead of justifying the optimization statically, a compiler can also synthesize preconditions that imply the correctness of the optimizations and are checked at the runtime of the program.

In this paper, we present a framework to collect, generalize, and simplify assumptions based on Presburger arithmetic. We introduce different assumptions necessary to enable a variety of complex loop transformations and derive a (close to) minimal set of preconditions to validate them at runtime. Our evaluation shows that the runtime verification introduces negligible overhead and that the assumptions we propose almost always hold true. On a large benchmark set including SPEC and NPB our technique increases the number of modeled non-trivial loop nests by a factor of 3.9×.

Categories and Subject Descriptors  D.3.4 [Programming Languages]: Compiler, Optimization

Keywords  Static Analysis; Presburger Precondition; Program Versioning; Polyhedral Model

1. Introduction

The polyhedral model has proven to be a very powerful vehicle for loop optimizations such as tiling, parallelization, and vectorization [2, 7, 8, 35, 36, 44]. It represents programs by convex polyhedra and leverages parametric integer programming techniques to analyze and transform them [18–20].

To be faithfully represented in the polyhedral model, a loop nest has to fulfill several strong requirements [19]. Amongst others, there must be no aliasing, all array subscripts must be affine, loop bounds must be loop invariant, and so on. Some of these constraints also impact the semantics of the programming language: loop counter arithmetic and subscript evaluation happens in \( \mathbb{Z} \) not in machine arithmetic. Arrays are for example truly multidimensional, thus no two different index vectors can access the same cell.

There is a trend to also use the polyhedral model on low-level languages such as C [11] and compiler intermediate representations (IRs) such as ORC/WRAP-IT [23], gcc/graphite [40], and LLVM/Polly [24]. Especially the semantics of IRs are often too low-level to fulfill all of the polyhedral model's requirements upfront. For example, LLVM-IR has no proper multidimensional arrays in the sense of Fortran, loop counter arithmetic might (depending on the input program and language) use modulo arithmetic, and aliasing rules are different due to the flat memory model. Some of these peculiarities can be worked around but usually this comes at an expense. Either significant increase in compile time, because the polyhedral representation becomes more complex, or less optimization potential, because of overapproximations on the program behavior [26], or both.

\[
\text{declare } \text{rhs}[\text{JMAX}][\text{IMAX}][5];
\]

\[
\text{for } (j = 0; j < \text{grid}[0] + 1; j++)
\]

\[
\text{for } (i = 0; i < \text{grid}[1] + 1; i++)
\]

\[
\text{for } (m = 0; m < 5; m++)
\]

\[
P: \text{rhs}[j][i][m] = /* ... */;
\]

Figure 1. Simplified excerpt of the `compute_rhs` function in the BT benchmark as provided in the C implementation of the NAS Parallel Benchmarks (NPB) [39].

The program in Figure 1 shows a simplified excerpt of the BT benchmark in the NAS parallel benchmark suite [39]. Several issues prevent the straightforward application of polyhedral techniques although existing work [32] has shown that it profits from such loop optimizations. To be polyhedrally representable it must satisfy three conditions.

1. The references to `grid` in the loop bounds must be loop invariant, i.e. these array cells must not be modified in the loop nest. This involves proving that this array does not alias with other arrays that are modified in the loop nest.
2. The loop bounds must not overflow since the polyhedral model is based on arithmetic in \( \mathbb{Z} \) not machine arithmetic.
3. The accesses must stay in-bounds with regards to the array allocation, i.e., \( i < \text{grid}[1] + 1 <= \text{IMAX} \). All these properties are notoriously hard to verify statically, if possible at all. However, we can hardly imagine a program run in which one of these requirements is violated. Hence, we are in the unsatisfactory situation that we know that these requirements are fulfilled for every program run of interest but we are unable to prove it. In this paper we solve this
Figure 2. The first two assumptions prevent integer overflow (Section 4.2) in the loop bounds, the third one out-of-bounds accesses (Section 4.5), and the last one ensures the absence of overlapping arrays (Section 4.6) and static control (Section 4.1).

In Figure 3 we list five situations with important semantic differences. In the polyhedral model there is no need to choose a type and a size for variables and arrays. The latter can span infinitely in each dimension and arithmetic operations can be performed in $\mathbb{Z}$. Consequently, the concept of out-of-bounds accesses or integer overflows does not apply. Additionally, each array can be placed in a disjunct part of the infinite memory, thus they are completely disjoint. This elides the possibility of aliasing, an integral part of low-level languages like C or LLVM-IR. Additionally, control flow has to be described statically and often needs to be bounded, thus dynamic control and (partially) infinite loops are prohibited.

These semantic mismatches can cause miscompilations, in case they are ignored. Nevertheless, it is common practice for a polyhedral optimizer to require (and sometimes document) that the input code is never called in situations that result in semantic mismatches. This requirement is not only hard to validate for programmers, but also hinders the automatic optimization of unvalidated source code.

<table>
<thead>
<tr>
<th>C</th>
<th>LLVM-IR</th>
<th>PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Referentially Transparent Expressions</td>
<td>(RT)</td>
<td>not-given</td>
</tr>
<tr>
<td>Expression Evaluation Semantics</td>
<td>(EE)</td>
<td>type-dependent</td>
</tr>
<tr>
<td>Always Bounded Loops</td>
<td>(BL)</td>
<td>no</td>
</tr>
<tr>
<td>Always In-bound Accesses</td>
<td>(IB)</td>
<td>sometimes$^1$</td>
</tr>
<tr>
<td>Aliasing Arrays</td>
<td>(AA)</td>
<td>possible</td>
</tr>
</tbody>
</table>

Figure 3. Semantics of C, LLVM-IR and the polyhedral model (PM) in different situations.

2.2 Architecture

Our approach allows to model programs that do not completely match the semantics of the polyhedral model by using...
ing optimistic assumptions to overcome the differences. Its overall design is depicted in Figure 4. We expand the traditional optimization flow of modeling a loop nest, deriving a transformation that is valid for all modeled program executions, and replacing the original code with an optimized version. Throughout the modeling and optimization phase we collect assumptions (Section 4) and generalize these assumptions to preconditions that must hold for the optimized code to reflect the original program behavior. These preconditions are then simplified (Section 5) and code is generated to ensure that the optimized loop nest is only executed if at runtime all preconditions are satisfied (Section 6). If not, it falls back to the original code.

3. Background

First, we provide background on affine expressions and Presburger sets before we introduce a simple core language that we then model using such sets.

3.1 Presburger Formulas and Sets

We use Presburger sets to describe properties and assumptions, as common operations on them are decidable. An \( n \)-dimensional Presburger set \( s \) is a parametric subset of \( \mathbb{Z}^n \). It is described by a Presburger formula that evaluates to \( \text{true} \) if a vector \( \vec{x} \in \mathbb{Z}^n \) is element of \( s \) and to \( \text{false} \) otherwise. A Presburger formula (Figure 5) is a boolean constant, a comparison between affine expressions, or a boolean combination of Presburger formulas. Presburger formulas also permit quantified variables. Affine expressions can reference local variables \( \langle \text{var} \rangle \) and unknown but constant parameters \( \langle \text{par} \rangle \).

We also use common extensions not described in Figure 5. An example two-dimensional set parameterized in \( N \) is \( D = \{ (d_0, d_1) \mid 0 \leq d_0 \leq d_1 < N \} \). An empty set is written as \( \{ \vec{d} \mid \text{false} \} \) and an universal one as \( \{ \vec{d} \mid \text{true} \} \). We use named Presburger sets which contain elements from different named spaces. The set \( \{ (B, (i, j)) \mid i < j \} \) contains elements named \( B \). A Presburger relation \( r \) is an element of \( \mathbb{Z}^n \times \mathbb{Z}^m \) and can be written as \( r = \{ (i_0, i_1) \rightarrow (o_0) : i_0 + i_1 \leq o_0 \} \). Presburger sets and relations are closed under set operations such as union, intersection, and difference. Sets can also be projected onto the parameter subspace, denoted as \( \pi_P(\cdot) \), which eliminates all variables \( \langle \text{var} \rangle \). The resulting set depends only on parameters \( \langle \text{par} \rangle \) and is empty for a given parameter valuation, \( \text{iff} \) the original set is empty for the same parameter valuation, i.e., \( \pi_P(D) = \{ 0 \leq N \} \). The operation \( r^{-1} \) denotes the inverse relation, thus interchanges domain and range. The image of a relation \( r \) under a set \( s \) is written as \( r(s) : = \{ t \mid \exists s \in S. (s, t) \in r \} \). We denote the complement of a set \( s \) as \( \neg s \).

3.2 Core Language

To illustrate code examples we introduce a core language (Figure 6) which is an extended version of Feautrier’s SCOP language [19]. We permit array reads in expressions, thus as part of base pointers, offset expressions, and control conditions. This allows indirect array access as well as dynamic control structures. The former is a common byproduct of arrays aggregated in a structure or class, the latter is often used to deal with variable sized arrays, e.g., the loop bounds are loaded dynamically from member or global variables. Additionally, we do not assume in-bounds array accesses.

### Figure 5. Affine expressions and Presburger formulas. Multiplication with a constant is reduced to repeated additions.

### Figure 6. Grammar for our core language.

While this language is otherwise tailored towards the use in polyhedral tools, it still allows to argue about the semantic differences of the polyhedral model and various real-world programming languages. The \( \langle \text{acc} \rangle \) rule describes accesses to an array with a possibly multi-dimensional offset. An expression \( \langle \text{exp} \rangle \) is an affine value (ref. \( \langle \text{aff} \rangle \) Figure 5) that also permits array reads as sub-expressions. Loop exit conditions \( \langle \text{cmp} \rangle \) are comparisons of two expressions and logical com-
bimations thereof. The final rule \( \langle stmt \rangle \) defines a statement, the top-level entity of the language. A statement can either be a declaration of an array, the assignment to an array location, a sequence of statements, a loop or a conditional.

<table>
<thead>
<tr>
<th>for ( i = 0; i &lt; N; i += 1 )</th>
<th>( \langle exp \rangle )</th>
<th>( \langle var \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle var \rangle )</td>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
</tr>
<tr>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
</tr>
<tr>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
</tr>
<tr>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
</tr>
<tr>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
</tr>
<tr>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
<td>( \langle exp \rangle )</td>
</tr>
</tbody>
</table>

Figure 7. Simple core language loop nest.

3.3 Polyhedral Representation of Programs

The polyhedral model is a well known mathematical program abstraction based on Presburger sets [21]. It allows to reason about control flow and memory dependences in static control programs (SCoPs [19]) with maximal precision. With the exception of array reads in expressions \( \langle exp \rangle \), core language programs can be natively translated into the polyhedral model. The iteration space \( I_{\theta} \) (aka. domain) of a program statement \( S \) is represented as a parametric subset of \( \mathbb{Z}^{k} \), where \( k \) denotes the number of loops surrounding \( S \). Each vector in \( I_{\theta} \) describes the values of the surrounding loop iteration variables for a dynamic execution of \( S \). The iteration domain of statement \( \mathcal{F} \) in Figure 7 can be written as

\[ I_{\mathcal{F}} = \{(i,j) | 0 \leq i < N \land 0 \leq j < M\}. \]

The individual array accesses in a statement \( S \) are modeled by a named integer relation \( \mathcal{A}_{\theta} \) that relates each dynamic instance of \( S \) to the array elements it accesses. In this context, the named spaces are used to distinguish between accesses to different arrays. The accesses of statement \( \mathcal{F} \) in Figure 7 are for example described by the relation

\[ \mathcal{A}_{\mathcal{F}} = \{(i,j) \rightarrow (A,(j,i)) \cup \{(i,j) \rightarrow (B,(i,j))\}. \]

4. Optimistic Assumptions

This section introduces the optimistic assumptions that are necessary for applicable and sound polyhedral modeling and optimization. Some of them are, usually in simpler forms, used in existing compilers, but have been generalized in this work. Others are, to the best of our knowledge, new or have not yet been formalized in this way. We use core language examples to illustrate the semantic differences between the polyhedral model and real-world programming languages and thereby motivate the need for optimistic assumptions.

4.1 Referential Transparent Expressions

Expressions \( \langle exp \rangle \) in the core language are similar to affine functions \( \langle aff \rangle \), but also allow array reads. While affine functions can be naturally represented in the polyhedral model, expressions containing reads cannot as they are not referentially transparent. If these non-pure expressions are used in control conditions the control flow is not static but data-dependent. If they are used in array subscripts, the access is data-dependent. To represent loops with data-dependent control or accesses, we optimistically assume expressions to behave as if they were static, thus not data-dependent but referentially transparent. As a result, the code shown in Figure 8 is represented as if the accesses to the UB and Idx arrays have been hoisted out of the loop. This is correct if the array offsets are invariant and the corresponding memory location is not modified. An offset is invariant if it does not contain loop variables \( \langle var \rangle \) and all sub-expressions are referentially transparent too. In order to determine if a potentially invariant read is overwritten, we first compute the set of all written locations \( \mathcal{W} \). To this end, the access relation of each array write is applied to the iteration domain of the surrounding statement. \( \mathcal{W} \) is then the union of the results.

\[ \text{for } (i = 0; i < UB[0]; i += 1) \]
\[ S: A[\text{Idx}[0] + i] += B[i]; \]

Figure 8. Invariant memory accesses that can be modeled as parameters in the domain of \( S \) and the access function of \( A \).

As illustrated in Figure 8, it is important to note that the values of the assumed invariant reads, i.e., \text{Idx}[0] and \text{UB}[0], affect the set \( \mathcal{W} \). If we denote their parametric values as \text{Idx}0 and \text{UB}0, then \( \mathcal{W} \) can be written as

\[ \mathcal{W} = \{(A, (\text{Idx}0 + i)) | 0 \leq i < \text{UB}0\}. \]

While \( \mathcal{W} \) is used to reason about the invariance of the assumed invariant reads, it also depends on their runtime values. Thus, it is generally not possible to reason about referentially transparent expressions only at compile time but runtime checks are needed to verify the assumptions. In order to determine the access relations and the iteration domains in the first place, we use the polyhedral representation of the region that can be build under the assumption that data-dependent control flow behaves as if it was static. This means to treat assumed invariant reads as parameters \( \langle \text{par} \rangle \).

Given the set \( \mathcal{W} \), one has to check that all reads \( r \) that have been assumed to be invariant actually are. We denote the access function of \( r \) as \( \mathcal{A}_{r} \) and the statement containing \( r \) as \( \mathcal{R} \). It remains to test if the read location \( \mathcal{A}_{r}(I_{\theta}) \) is contained in \( \mathcal{W} \) or not. If it is not, \( r \) is invariant, otherwise there exists at least one parameter combination for which \( \mathcal{A}_{r}(I_{\theta}) \) is written inside the analyzed region. Nevertheless, the optimistically built polyhedral representation remains sound under the assumptions these parameter valuations do not occur at runtime. The set of parameter valuations that do not cause a write to the location \( \mathcal{A}_{r}(I_{\theta}) \) is

\[ \Lambda_{RT}(r) = \pi_{\rho}(\neg(\mathcal{W} \cap \mathcal{A}_{r}(I_{\theta}))). \]

The intersection of \( \Lambda_{RT} \) for all assumed invariant reads (AIR) describes the valid parameter combinations under which all expressions are referentially transparent. Thus, the referentially transparent assumption is defined as

\[ \Lambda_{RT} = \bigcap_{r \in AIR} \Lambda_{RT}(r). \]

4.2 Expression Evaluation Semantics

The polyhedral model is a mathematical program abstraction which evaluates expressions in \( \mathbb{Z} \). We denote this expression evaluation semantics as \text{Precise}. However, programming languages like Java, Julia, C/C++ [15], and LLVM-IR impose more machine dependent semantics on expression evaluation. The two most common ones are \text{Wrapping}, thus

\[ \text{for } (i = 0; i < UB[0]; i += 1) \]
\[ S: A[\text{Idx}[0] + i] += B[i]; \]
the evaluation modulo \( m = 2^n \) for \( n \)-bit expressions, and \textbf{Error} which causes undefined behaviour if the result of \textit{Precise} and \textit{Wrapping} evaluation differs. Note that \textit{Precise} semantics subsume \textit{Error} semantics, but not \textit{Wrapping} semantics.

In order for the polyhedral model to represent the input correctly, it is necessary to represent the evaluation semantics as well. While it is possible to express \textit{Wrapping} semantics in Presburger arithmetic [45], practice shows that it has a vastly negative effect on compile time as well as runtime of the generated code (Section 7.2). The former is caused by the additional existentially quantified dimensions that modulo expressions can introduce, the latter by additional dependences that are only present in case of wrapping (Figure 9). While integer wrapping rarely occurs when executing the programs commonly analysed by polyhedral tools, an automatic approach used on general purpose code should never silently mis-compile programs for corner-case inputs.

To represent possibly wrapping computations in an efficient way, we optimistically use \textit{Precise} semantics. This is sound for parameter valuations that do not cause any expression to wrap. We denote these parameter valuations as \( \text{exp} \). To compute them, each expression \( e \in \{\exp\} \) of the input program is translated twice to the polyhedral model. First with \textit{Precise} semantics and then with \textit{Wrapping} semantics. We use \( \lfloor e \rfloor_z \) to express the former translation and \( \lfloor e \rfloor_{Z/mZ} \) for the latter. Both translate \( e \) to a function in the surrounding iteration variables. Assuming \( e \) is part of statement \( s \) and surrounded by \( k \) loops, we can compute \( I_W(e) \), the set of all iterations for which \( e \) would wrap:

\[
I_W(e) = \{ (i) \mid i \in Z^k \land \lfloor e \rfloor_z(i) \neq \lfloor e \rfloor_{Z/mZ}(i) \}.
\]

To restrict it to actually executed iterations, \( I_W(e) \) is intersected with the iteration space \( I_s \) of the statement \( s \):

\[
I_W(e) = I_W(e) \cap I_s.
\]

The negated projection of \( I_W(e) \) onto the parameter subspace describes the parameter evaluations under which the evaluation of \( e \) with \textit{Precise} semantics is equal to the evaluation with \textit{Wrapping} semantics, thus the expression evaluation assumptions \( \Lambda_{EE}(e) \) for \( e \). The intersection of all \( \Lambda_{EE}(e) \) yields the preconditions \( \Lambda_{EE} \) that ensure the absence of wrapping in all control and access expressions:

\[
\Lambda_{EE} = \bigcap_{e \in \{\exp\}} \Lambda_{EE}(e) = \bigcap_{e \in \{\exp\}} \neg \pi_p(I_W(e)).
\]

It is important to note that the optimistically generated polyhedral representation of the input program is not necessarily sufficient to compute \( \Lambda_{EE} \). Due to the referential transparency of expressions in the polyhedral model, it is possible that values are replaced by their definition, thus altering the domain under which an expression is evaluated. In Figure 10 two equivalent programs are shown if expressions are evaluated with \textit{Precise} semantics but not necessary with \textit{Wrapping} semantics. While the polyhedral representation of Figure 10a and Figure 10b can be equal (Figure 10c) the former might exhibit a wrapping increment expression while the latter does not. Consequently, it is necessary to utilize the textual expression and the textual domain to compute \( \Lambda_{EE} \), not a polyhedral representations thereof.

![Figure 9. Loop with dependences only if Wrapping semantics are used. Assuming \( i \) to be an 8-bit unsigned value, loop-carried dependences are then present if \( N = 2^7 = 128 \).](image)

To represent possibly wrapping computations in an efficient way, we optimistically use \textit{Precise} semantics. This is sound for parameter valuations that do not cause any expression to wrap. We denote these parameter valuations as \( \text{exp} \). To compute them, each expression \( e \in \{\exp\} \) of the input program is translated twice to the polyhedral model. First with \textit{Precise} semantics and then with \textit{Wrapping} semantics. We use \( \lfloor e \rfloor_z \) to express the former translation and \( \lfloor e \rfloor_{Z/mZ} \) for the latter. Both translate \( e \) to a function in the surrounding iteration variables. Assuming \( e \) is part of statement \( s \) and surrounded by \( k \) loops, we can compute \( I_W(e) \), the set of all iterations for which \( e \) would wrap:

\[
I_W(e) = \{ (i) \mid i \in Z^k \land \lfloor e \rfloor_z(i) \neq \lfloor e \rfloor_{Z/mZ}(i) \}.
\]

To restrict it to actually executed iterations, \( I_W(e) \) is intersected with the iteration space \( I_s \) of the statement \( s \):

\[
I_W(e) = I_W(e) \cap I_s.
\]

The negated projection of \( I_W(e) \) onto the parameter subspace describes the parameter evaluations under which the evaluation of \( e \) with \textit{Precise} semantics is equal to the evaluation with \textit{Wrapping} semantics, thus the expression evaluation assumptions \( \Lambda_{EE}(e) \) for \( e \). The intersection of all \( \Lambda_{EE}(e) \) yields the preconditions \( \Lambda_{EE} \) that ensure the absence of wrapping in all control and access expressions:

\[
\Lambda_{EE} = \bigcap_{e \in \{\exp\}} \Lambda_{EE}(e) = \bigcap_{e \in \{\exp\}} \neg \pi_p(I_W(e)).
\]

It is important to note that the optimistically generated polyhedral representation of the input program is not necessarily sufficient to compute \( \Lambda_{EE} \). Due to the referential transparency of expressions in the polyhedral model, it is possible that values are replaced by their definition, thus altering the domain under which an expression is evaluated. In Figure 10 two equivalent programs are shown if expressions are evaluated with \textit{Precise} semantics but not necessary with \textit{Wrapping} semantics. While the polyhedral representation of Figure 10a and Figure 10b can be equal (Figure 10c) the former might exhibit a wrapping increment expression while the latter does not. Consequently, it is necessary to utilize the textual expression and the textual domain to compute \( \Lambda_{EE} \), not a polyhedral representations thereof.

![Figure 10. Loops with equal polyhedral representation but different wrapping behaviour.](image)

**4.3 Possibly Unbounded Loops**

Possibly unbounded loops are an implementation artifact that can cause complex, partially unbounded iteration domains and thereby compile time hazards. In practice, possibly unbounded loops are often caused by parametric loop bounds with an equality exit condition, thus \( = \) or \( == \). Such exit conditions are used by programmers but also introduced by canonicalizing program transformations. An example loop that is possibly unbounded is shown in Figure 11². For \( \text{LB} > \text{UB} \) the iteration domain of statement \( s \) is unbounded. Thus, in the polyhedral representation this parameter valuation would cause an infinite loop and thereby compile time hazards while its occurrence in practice would either result in an error or render optimizations redundant.

![Figure 11. Partially unbounded domain for \( \text{LB} > \text{UB} \). In order to keep the iteration domains bounded and concise while still being able to handle loops with a potentially unbounded number of iterations we generate preconditions that prevent unbounded loops statically. Using the example above we first bring the iteration domain](image)

\[
I_s = \{ \{ i \mid \text{LB} \leq i < \text{UB} \} \} \lor \{ \{ i \mid \text{UB} < \text{LB} \leq i \} \}
\]

into disjunctive normal form and identify all clauses that do not bound all loop iteration variables properly. In this example the first disjunct provides proper bounds for \( i \) while the second does not provide an upper bound. We denote the set of unbounded clauses in the domain \( I_s \) as \( I_s^\infty \). The negated projection of \( I_s^\infty \) onto the parameter space yields the bounded loops assumption \( \Lambda_{BL}(s) \). All parameter valuations that would cause an unbounded number of loop it-
izations for statement $S$ are precluded by $\Lambda_{BL}(S)$. Thus, the intersection of $\Lambda_{BL}(S)$ over all statements prevents unbounded domains all together:

$$\Lambda_{BL} = \bigcap_S \Lambda_{BL}(S) = \bigcap_S \neg \pi_{\rho}(I_S^{\infty}).$$

Note that for nested loops with dependent conditionals, as illustrated in Figure 12, prior dimensions are assumed to be bounded. Hence, the constraints $0 \leq j < i$ suffice as bound for the loop iteration variable $j$ in the domain of $S$.

```
for (i = 0; ...; ...)  
  for (j = 0; j < i; ...)  
    S: A[i][j] = B[i][j];
```

**Figure 12.** Generic nested loop with dependent conditionals.

### 4.4 Accesses to Constant-Size Arrays Are In-bounds

Functions that work on multi-dimensional arrays of fixed size often do not provide sufficient information to prove that all memory access subscript expressions remain within bounds. A common reason for this is the use of parametric array bounds, which appear either just due to inconsistent code or, as illustrated in Figure 13, due to code that works only on sub-arrays. In some languages multi-dimensional out-of-bound accesses are disallowed or result in runtime errors. Other languages linearize multi-dimensional array accesses and treat them as one-dimensional ones. Such accesses remain valid even if not all subscript expressions remain within the dimension bounds of the multi-dimensional array as long as the location accessed is valid. While LLVM-IR retains information about the multi-dimensionality of accesses for arrays with constant sized dimensions, there is no guarantee such accesses remain within bounds: “Analysis passes which wish to understand array indexing should not assume that the static array type bounds are respected”.

```
declare A[1024][1024];  
declare B[1024][1024];  
for (i = start; i < start + num; i++)  
  for (j = 0; j < i; j++)  
    S: A[i][j] = B[i][j];
```

**Figure 13.** Parametric accesses to constant-sized arrays.

Accesses to multi-dimensional arrays of constant size that have affine subscripts in each dimension can be equivalently expressed as affine one-dimensional accesses. Therefore, existing integer programming based dependence analysis techniques [19, 37] can be used to compute precise results. However, out-of-bound memory accesses can introduce spurious data dependences that prevent otherwise legal program transformations. To illustrate the problem we consider the example in Figure 13. The set

$$I_S = \{(i, j) | \text{start} \leq i < \text{start} + \text{num} \land \text{start} \leq j < \text{start} + \text{num}\}$$

describes all iterations of $S$ and the relation

$$A_S = \{(i, j) \rightarrow (A, (1024i + j))\} \cup$$

$$\{(i, j) \rightarrow (B, (1024i + j))\}$$

describes the accesses performed. The source code suggests that the code is dependence free and that transformations such as loop interchange are valid. However, if the language allows for out-of-bounds accesses in the individual offset expressions, e.g., as LLVM-IR does due to the implicit linearization, the set of data dependences is not empty:

$$\{(i, j) \rightarrow (i + 1, j - 1024)\}  
  \text{start} \leq i < \text{start} + \text{num} - 1 \land  
  1024 + \text{start} \leq j < \text{start} + \text{num}\}$$

For values of $j$ that are larger than $1024 + \text{start}$, there is a data dependence from iteration $(i, j)$ to the later iteration $(i + 1, j - 1024)$ caused by an out-of-bound memory access. If we consider only the values of $j$ that do not cause out-of-bound accesses, the set of data dependences is empty.

As out-of-bound accesses are valid, but uncommon, we can optimistically assume they never happen, thereby avoiding the spurious dependences. To this end, we first define constraints to describe out-of-bound memory locations and then compute the iterations that access such locations. For an $N$-dimensional array, where each dimension $i$ has size $s_i$, the out-of-bound memory locations are described as

$$M_{Out} = \{(e_i) | \bigvee_{i=1..(N-1)} (e_i < 0 \lor s_i \leq e_i)\}.$$

To obtain the set of iterations $I_{Out}$ that perform an out-of-bound access, we apply for each statement $S$ the reverse access relation $A_S^{-1}$ to the out-of-bound access description and restrict the result to the statement domain $I_S$, thus:

$$I_{Out} = \bigcup_S (I_S \cap A_S^{-1}(M_{Out}))$$

The projection of $I_{Out}$ onto the parameter subspace yields a description of all parameter combinations that trigger at least one out-of-bound access. Taking the complement, we derive the assumptions that ensure in-bounds accesses:

$$\Lambda_{IB} = \neg \pi_{\rho}(I_{Out})$$

### 4.5 Accesses to Parametric-Size Arrays Are In-bounds

Accesses to multi-dimensional arrays of parametric size are, similar to their fixed-sized counterparts, modeled in the compiler IR as one-dimensional accesses. However, even if the individual subscript expressions were affine (e.g., $A[i][j]$), the linearized result is commonly a polynomial expression (e.g., $A[i * n + j]$) which cannot be analyzed with ILP-based techniques. Grosser et. al [26] presented a delinearization approach that guesses possible multi-dimensional array accesses by looking for non-affine monomials in the polynomial access functions. In many cases, the correctness of this delinearization is not statically provable, but an assumption can be constructed that ensures the correctness. Our framework is used to keep track.
of these delinearization assumptions, to simplify them with respect to other (independently taken) assumptions, and to emit optimized runtime checks. In the evaluation (Section 7) we record delinearization assumptions with the in-bounds assumptions as $\Lambda_{IB}$.

4.6 Arrays Do Not Alias (Overlap)

Alves et al. [4] presented an approach to rule out array aliasing at runtime that utilizes the optimistic assumption framework presented in this work. Their runtime check verifies that two array regions which are accessed via different base pointers are not overlapping, thus not aliasing. The access ranges for all possible overlapping arrays are computed in the same way as the set of written locations $W$ in Section 4.1. As a consequence of our extensions, the access ranges can, similar to $W$, be dependent on the values of assumed invariant reads and the absence of overflows. Hence, only the combination of alias assumptions $\Lambda_{AA}$ and referentially transparent assumption $\Lambda_{RT}$ allows to handle loops with assumed invariant loads which might alias other arrays.

To model the example in Figure 1 one has to assume the first two elements of grid are not overwritten. Aliasing checks need to argue about the accessed memory regions, thus they depend on the loop bounds that are not static. At the same time one cannot assume the loop bounds to be invariant if any aliasing access could dynamically change them. Only by assuming and verifying both properties simultaneously, a correct model can be built.

5. Efficient Assumptions

Handling assumptions efficiently is important to minimize compile time and to ensure their fast evaluation at runtime. The number of assumptions inevitable increases with program size, but their cost is often more impacted by the kind and representation of the assumptions. We exploit flexibility in the assumptions we take to obtain simpler Presburger assumptions only for in-bounds assumptions $\Lambda_{IB}$. We use undefined behavior, e.g., to not generate expression evaluation assumptions $\Lambda_{EE}$ if a computation is guaranteed to not overflow. As the relaxed type system and memory model of compiler IRs is often insufficient to model necessary language semantics, we use annotations to carry over missing information. In case of out-of-bound accesses to fixed-sized arrays, which are undefined in C/C++ but not in LLVM-IR (Section 4.4), we emit annotations in the C/C++ frontend that guarantee in-bounds accesses, thus allow to omit in-bounds assumptions.

Positive Assumptions vs. Restrictions: Assumptions can be modeled as set of valid parameter configurations (positive assumptions) or as set of invalid parameter configurations (restrictions) and this choice significantly impacts the representation efficiency. In Section 4, we introduced all assumptions as positive assumptions. However, depending on how Presburger sets are represented, restrictions can be advantageous. Polly relies on isl [42], which uses a disjunctive normal form (DNF) as canonical representation. When collecting positive assumptions, new constraints are added by intersection. When collecting restrictions, new constraints are added by computing the union. Intersecting is fast for single convex polyhedra, where it corresponds to appending constraints. However, when individual assumptions are represented by a union of convex polyhedra, computing the DNF of an intersection can increase the size of its representation drastically due to the distributive property. In contrast, restrictions grow linearly. In our implementation we use positive assumptions only for in-bounds assumptions $\Lambda_{IB}$ and restrictions otherwise.

Conservative Over-Approximation: In certain cases conservative approximations of assumptions allow for more concise Presburger sets without observable disadvantages in practice. For example, a simple non-uniform stride (Figure 14a) can cause a complicated runtime alias check (Figure 14b) which can be conservatively simplified (Figure 14c). Since especially existentially quantified dimensions, which often arise from non-uniform strides or modulo expressions, have shown to complicate assumptions, we conservatively approximate assumptions by projecting out such dimensions.

6. Runtime Check Generation

So far we have shown how to take, combine, and simplify assumptions as preconditions for efficient, sound, and optimistic loop optimizations. At runtime, these preconditions are evaluated to determine if it is valid to execute the optimistically optimized loop nest or if the conservatively optimized one needs to be used. In either case, it is crucial that the code that is used to evaluate these preconditions correctly
While CPUs use
continue the evaluation of the runtime check code. After the
detection for arithmetic operations, this can be implemented
check code. Especially on hardware with build-in overflow
itation semantics we track potential overflows in the runtime
To bridge the gap between the different expression evalua-
Figure 14. Complicated and conservatively simplified run-
time alias checks for a simple loop with non-uniform stride.
implies their semantics. In Figure 2b we illustrated how as-
sumptions can be generalized to the whole region. However,
code for runtime checks cannot be simply generated for the
collected and simplified assumption. Two additional chal-
enges arise in order for the runtime check code to be a, pos-
sibly weaker but sound, precondition.
1. Machines use Wrapping\(^4\) semantics (ref. Section 4.2) to
evaluate expressions, not the Precise semantics that is
used to combine and simplify the assumptions in the poly-
hedral model. This discrepancy can cause subtle errors,
especially in the context of expression evaluation assump-
tions \(\Lambda_{EE}\) that may contain large constants.
2. Preconditions can reference assumed invariant reads
(ref. Section 4.1) as part of parameters in the polyhedral
model. These reads have to be “pre-loaded” to make their
values available during the runtime check generation.

Algorithm 1: Runtime check generation.

\underline{Input}: an affine function \(q \in \langle \text{aff} \rangle\)
\underline{Output}: code that computes \(q\) or signals a failure.

\begin{algorithmic}[1]
\Function{generateAff}{\(q\)}
\Switch{\(q\)}
\Case{c} \Return \(c\); \quad \Comment{\(c \in \langle \text{int} \rangle\)}
\Case{v} \Return \(v\); \quad \Comment{\(v \in \langle \text{var} \rangle\)}
\Case{p} \Return \(p\); \quad \Comment{\(p \in \langle \text{par} \rangle\)}
\EndSwitch
\EndFunction
\end{algorithmic}

To bridge the gap between the different expression evaluation
semantics we track potential overflows in the runtime
check code. Especially on hardware with build-in overflow
detection for arithmetic operations, this can be implemented
efficiently. For the example in Figure 2b this means that after
each addition we check explicitly for an overflow before we
continue the evaluation of the runtime check code. After the
first overflow the runtime check fails and the conservatively
optimized code is executed.

To make the values of assumed invariant reads available
in the runtime check, they have to be hoisted in front of the
analyzed region. While this is generally possible, it is im-
portant that an invariant read should only be pre-loaded un-
der the condition that the memory location can be safely ac-
cessed. For the example in Figure 2b this means that the ac-
cess to \(\text{grid}[1]\) must not be performed if \(\text{grid}[0] < 0\).

The two mutually recursive Algorithms 1 and 2 illus-
 trata how we extended code generation for Presburger for-
mula [27] to tackle the additional challenges that come with
sound and efficient runtime check generation. While the first
three cases shown in Algorithm 1 do conceptually not dif-
fer from common code generation for affine functions, the
last case (line 7) was extended. Additional code that detects
a potential overflow at runtime is emitted after each poten-
tially overflowing arithmetic instruction. In case an overflow
occurred, thus a failure is signaled, the conservatively opti-
mized code version has to be executed. It is important not to
cause any side-effect after a problem in the runtime check
has been detected. To this end, pre-loaded assumed invariant
reads have to be guarded explicitly if it cannot be shown that
the memory can be accessed unconditionally.

Algorithm 2: Parameter generation for runtime checks.

\underline{Input}: a parameter \(p \in \langle \text{par} \rangle\) that might reference
assumed invariant reads
\underline{Output}: code that computes \(q\) or signals a failure.

\begin{algorithmic}[1]
\Function{generateParameterOrArrayRead}{\(p\)}
\ForEach{array read \(a\) in \(p\)} \quad \Comment{\(a \in \langle \text{acc} \rangle\)}
\If{isPotentiallyUndefinedAccess\(a\)}
\State \(I_a \leftarrow \text{getDomainForAccess}(a)\); \quad \Comment{\(I_a \in \langle \text{int} \rangle\)}
\State \(\text{generateFailureIfEmpty}(I_a)\); \quad \Comment{\(I_a \in \langle \text{par} \rangle\)}
\State \(bp \leftarrow \text{getBasePointerAff}(a)\); \quad \Comment{\(bp \in \langle \text{aff} \rangle\)}
\State \(\text{addr} \leftarrow \text{generateAff}(bp)\); \quad \Comment{\(\text{addr} \in \langle \text{var} \rangle\)}
\EndIf
\EndFor
\If{offset expression \(e\) in \(p\)} \quad \Comment{\(e \in \langle \text{exp} \rangle\)}
\State \(\text{noWrap} \leftarrow \text{generateAssumptions}(\Lambda_{EE}(e))\); \quad \Comment{\(\Lambda_{EE}(e) \in \langle \text{par} \rangle\)}
\State \(\text{generateFailureIfFalse}(	ext{noWrap})\); \quad \Comment{\(\text{noWrap} \in \langle \text{bool} \rangle\)}
\State \(e_q \leftarrow \text{getExpressionAff}(e)\); \quad \Comment{\(e_q \in \langle \text{aff} \rangle\)}
\State \(\text{addr} \leftarrow \text{addr} + \text{getAff}(e_q)\); \quad \Comment{\(\text{addr} \in \langle \text{var} \rangle\)}
\State \(I_l \leftarrow \text{generateLoad}(\text{addr})\); \quad \Comment{\(I_l \in \langle \text{var} \rangle\)}
\EndIf
\State \(\text{replace} a \text{ with } l \text{ in all parameters and expressions};\)
\EndFunction
\end{algorithmic}
are pre-loaded first through the (mutual) recursion in line 7 and line 12. Finally, the expression evaluation assumptions \( \Lambda_{EE} \) for each offset expression have to be checked prior to the access. If one of them is violated, it is not sound to perform the access, as there might be an integer overflow that was not represented correctly. Thus, the location accessed by the program might not be the one accessed in the model. The real location might not be invariant or might just be different from the one that would have been pre-loaded. In either case, the conservative optimized version has to be executed.

7. Evaluation

To evaluate the assumptions collection, the simplification, and the runtime check generation, we run Polly on the LLVM Test Suite, the NPB Suite, and the C/C++ benchmarks of the SPEC 2000 as well as 2006 benchmark suite. The evaluation is restricted to non-trivial regions, thus loop nests that contain at least two loops or two statements with memory accesses (both read and write) inside loops. This granularity is the finest one could expect polyhedral optimizations to be effective, thus transformations like loop interchange or loop fusion/fission to be applicable. All performance numbers are generated with an Intel(R) Xeon(R) E3-1225. We used the default input size for the LLVM Test Suite, train input for SPEC, and the W input class for NPB.

<table>
<thead>
<tr>
<th>SPEC 2006</th>
<th>SPEC 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) w/ ( \Lambda )</td>
<td>(b) w/o ( \Lambda )</td>
</tr>
<tr>
<td>#S 191</td>
<td>35</td>
</tr>
<tr>
<td>#D 34</td>
<td>12</td>
</tr>
<tr>
<td>#E 5.2M</td>
<td>61k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NPB</th>
<th>LLVM Test Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) w/ ( \Lambda )</td>
<td>(b) w/o ( \Lambda )</td>
</tr>
<tr>
<td>#S 50</td>
<td>2</td>
</tr>
<tr>
<td>#D 41</td>
<td>1</td>
</tr>
<tr>
<td>#E 214k</td>
<td>48k</td>
</tr>
</tbody>
</table>

Figure 15. #S denotes the number of analyzed non-trivial loop nests (a) and how many had statically infeasible assumptions (b). #D shows how many of these were executed by the test suite (a) and how many violated the assumptions (b). #E denotes how often they were executed (a) and how often they violated an assumption (b).

7.1 Applicability

Figure 15 presents statistics about the applicability of our approach (w/ \( \Lambda \)) compared to Polly without assumptions (w/o \( \Lambda \)). First #S, gives the number of non-trivial regions that were analyzed (a) together with the number of regions for which infeasible assumptions were taken (b). As an example, Polly analyzes 191 non-trivial regions in SPEC 2006. Out of which 35 do not require any assumptions to be taken and 191 − 35 = 156 do. However, not all 156 regions will actually be optimized. For 89 regions statically infeasible assumptions were taken, thus the regions were dismissed during the modeling. Summarized, optimistic assumptions allow to optimize almost three times as many non-trivial regions in the SPEC 2006 benchmarks. Line #D shows how many of these distinct loop nests were executed during a run of the test suite (a) and how many of them violated the assumptions in at least one execution (b). In terms of dynamic total (#E), SPEC 2006 executed the optimized regions 5.2 million times (a) and in 16k of these executions the runtime checks did not hold (b). All but 6 dynamic mis-speculations were caused by a single loop nest in the 403.gcc benchmark. Similarly we can identify one loop nest in each of the benchmark suites to account for 82% of all runtime check failures.

<table>
<thead>
<tr>
<th>SPEC 2006</th>
<th>SPEC 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) w/ ( \Lambda )</td>
<td>(b) w/o ( \Lambda )</td>
</tr>
<tr>
<td>( \Lambda_{IB} )</td>
<td>5</td>
</tr>
<tr>
<td>( \Lambda_{EE} )</td>
<td>611</td>
</tr>
<tr>
<td>( \Lambda_{BL} )</td>
<td>42</td>
</tr>
<tr>
<td>( \Lambda_{AA} )</td>
<td>132</td>
</tr>
<tr>
<td>( \Lambda_{RT} )</td>
<td>553</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NPB</th>
<th>LLVM Test Suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) w/ ( \Lambda )</td>
<td>(b) w/o ( \Lambda )</td>
</tr>
<tr>
<td>( \Lambda_{IB} )</td>
<td>1021</td>
</tr>
<tr>
<td>( \Lambda_{EE} )</td>
<td>773</td>
</tr>
<tr>
<td>( \Lambda_{BL} )</td>
<td>0</td>
</tr>
<tr>
<td>( \Lambda_{AA} )</td>
<td>14</td>
</tr>
<tr>
<td>( \Lambda_{RT} )</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 16. The \( \Lambda_\ast \) rows show how many non-trivial assumptions were taken (a) and not implied by prior ones (b).

The \( \Lambda_\ast \) rows in Figure 16 show how often assumptions were taken (a) and then how often they were not already implied by prior ones (b). Though, the order in which the assumptions are taken influences the second number, we believe it is interesting to see how often assumptions are already implied, thus have no impact on the runtime check.

7.2 Modeling Choices, Simplification, and Versioning

Especially the expression evaluation assumptions \( \Lambda_{EE} \) and the bounded loop assumptions \( \Lambda_{BL} \) are alternatives to an otherwise complex and costly representation. While the latter are currently required in the optimization pipeline, the former can be avoided by explicitly modeling Wrapping semantics. However, the compile time will increase for various benchmarks between 3% and 3k\%, causing a timeout after 500s of compile time for 8 of them.

Simplifications (Section 5) generally reduce compile time. However, due to heuristics which exploit the constraint representation and newly exposed optimization opportunities, compile time increases can be observed in certain situations. The most important change we see is the elimination of compile time hazards. An example is the Linpack [17] benchmark. It is optimized in less than 3 seconds with assumption simplifications but requires more than 500 seconds without.
If the assumptions are not taken but the optimistically optimized version is unconditionally executed, we see overall compile time improvements of up to 24%. The runtime decrease without runtime checks stays below 4% of the overall execution time.

7.3 Sound and Automatic Polyhedral Optimization

Our assumptions allow to apply existing polyhedral approaches [4, 25, 32, 33] in a sound and automatic way on low-level code without the need for manual pre-processing. For our motivating example, the compute rhs function of the BT benchmark from the NPB suite (excerpt shown in Figure 1 and 2), this would be an $6 \times$ fold speedup with 8 threads reported by Mehta and Yew [32].

In addition, we can observe speedups in general purpose codes. The most interesting case is the P7Viterbi function of the 456.hmmer benchmark in the SPEC 2006 benchmark suite. The innermost loop in this function cannot be vectorized by LLVM due to the loop carried dependences induced in the middle part of the loop$^3$. However, the top as well as bottom perform independent computations that do not cause loop carried dependences. The loop distribution performed by Polly exposes the vectorization opportunity in the bottom part to LLVM, which reduces the total execution time of 456.hmmer (on the reference input) by 28% compared to clang−3.8 -O3.

Finally, the optimistic assumptions allow to optimize loop nests written in the programming style used by Julia or the boost: ublas C++ library. In both cases arrays (and matrices) are structures that contain not only the data but also their size. The latter is then dynamically loaded inside the loop nest, e.g., as upper bound for loops. This programming style causes data-dependent control flow (ref. Section 4.1), potential multidimensional out-of-bound accesses (ref. Section 4.5) as well as potentially aliasing accesses (ref. Section 4.6). Without our optimistic assumptions manual intervention is necessary for all programs written in this style.

8. Related Work

Optimistic assumptions are special preconditions, a topic well studied over the years [12, 14, 29]. Especially in the context of runtime check elimination for safe languages, several methods have been proposed [10, 22, 34, 38, 48]. These approaches generate an optimistic assumption, or precondition, to exclude out-of-bounds array accesses. In contrast, we employ them as a means to ensure a correct abstraction, simplify dependences, and to allow more optimization. Nevertheless, the two in-bounds related assumptions $\Lambda_{IB}$ share similarities with many of the algorithms and methods proposed in the literature: one of the oldest being by Cousot and Halbwachs [14]. With their abstract interpretation based on a relational domain, they can e.g., prove the absence of out-of-bounds accesses in classical SCoPs [19].

Integer overflows have been detected statically [13] as well as dynamically [15]. The work closest to our non-wrapping assumption $\Lambda_{EE}$ derives input filters to prevent integer overflows [31]. As they completely give up control constraints in favor of performance, we believe our assumptions could tighten and simplify their checks significantly.

The polyhedral extraction tool (PET) [45] might produce piecewise defined, partially unbounded iteration domains that are not easy to deal with and can cause compile time hazards. PET also explicitly models wrapping for unsigned integer which we have found to be expensive and not beneficial in practise. Alternatives [6] are not generally applicable for the loops of interest. In abstract interpretation, Urban and Miné [41] developed a termination analysis that implicitly derives bounded assumptions $\Lambda_{BL}$ for structured code.

Invariant code hoisting is a well known optimization [3]. However, we are not aware of any approach that optimistically hoists array reads in combination with dynamic alias checks as we do. Alternatively, control flow overapproximation [9, 33] can be used either in conjunction or as an approximative replacement. Though, for the latter, the optimisation potential will be limited. The delinearization and non-alias assumptions have already been discussed elsewhere [4, 26]. We integrate them into a general assumption framework.

LLVM [30] natively shares boolean assumptions between passes, but there is no simplification performed. Hoenicke et al. [29] used static analysis to identify statements for which the execution inevitably fails. While we currently skip optimizations if the needed assumptions are known-infeasible, we could similarly flag such regions as suspicious.

Lastly, we share ideas and problems with other runtime variant selection schemes [5, 16, 33, 46, 47], though we currently only generate all or nothing assumptions. Pradelle et al. [36] describe how to manually generate and dynamically select different program versions through polyhedral optimizations. Utilizing our assumption framework, it would be possible to automatically generate such optimized variants based on different assumptions made during scheduling.

9. Conclusion

In this work we present a set of optimistic assumptions that formally describe necessary and sufficient preconditions to optimize low-level code with polyhedral approaches. These assumptions are precise for programs with affine conditions and memory accesses and allow over-approximations for others. Our implementation automatically collects and simplifies all necessary assumptions to apply polyhedral optimizations on LLVM-IR programs in a sound and automatic fashion. The run-time checks that verify statically undecided assumptions dynamically are (close to) minimal and induce only little overhead. At the same time our simplifications reduce both compile and runtime significantly. Overall, this work enables complex and sound optimizations for general purpose code with unexpected corner cases.

Acknowledgements We thank Swissuniversities for support through the PASC initiative (ComPASC) and ARM Inc. for supporting Polly Labs.

---

$^3$ LLVM does not by default perform loop distribution and the available implementation works, in contrast to Polly, only on innermost loops.
References


the 2016 International Conference on Supercomputing, ICS ’16, pages 1:1–1:13, New York, NY, USA, 2016. ACM.


A. Artifact Description

A.1 Abstract
The work described in this paper has been fully implemented as an extension of the open source LLVM/Polly project and has been contributed to the Polly project repository. All it takes to test our implementation is a recent version of LLVM, Clang, and Polly.

Interactive scripts and a step-by-step description to reproduce the experiments and validate the implementation are available at:

github.com/jdoerfert/CGO17_ArtifactEvaluation

A.2 Software Versions
We used the software versions shown in Table 1 for the evaluation in Section 7.

<table>
<thead>
<tr>
<th>Software</th>
<th>Version (git/svn/release)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM</td>
<td>bdf16bd (svn: r288240)</td>
</tr>
<tr>
<td>Clang</td>
<td>1f955bd (svn: r288231)</td>
</tr>
<tr>
<td>Polly</td>
<td>b60757c (svn: r288521)</td>
</tr>
<tr>
<td>LLVM Test Suite</td>
<td>1d312ed (svn: r287194)</td>
</tr>
<tr>
<td>NPB</td>
<td>3.3 Serial C</td>
</tr>
<tr>
<td>SPEC 2006</td>
<td>1.1</td>
</tr>
<tr>
<td>SPEC 2000</td>
<td>1.3.1</td>
</tr>
</tbody>
</table>

Table 1. Software versions used for the evaluation.

A.3 How Delivered
We provide a docker image to ease the machine set up. Additionally, interactive python scripts download, build, and run the experiments. We also describe how to get, build, and run everything manually.

A.4 Hardware Dependencies
We recommend 40 GB of free disk space and at least 8 GB of main memory.

A.5 Software Dependencies
A C11/C++11 compatible compiler as well as common build tools (cmake, python2, virtuenv, git, grep, sed, ...).

A.6 Datasets
SPEC2000 and SPEC2006 have been used in our evaluation, but experiments can also be run on the openly available LLVM nightly test suite.

A.7 Installation
The installation is identical to the source installation of LLVM/Polly. The test environment may require some additional setup to be performed, but scripts are provided that automate these steps.

A.8 Experiment Workflow
Most experiments are compilations with enabled statistic collection. The data on the applicability and the effect of the proposed assumptions is then reported to the user and can be summarized using the provided scripts. Additionally compile time and runtime measurements can be run. The test environment (lint) that is used in our documentation allows to run both automatically. It also displays the results through a local web server.

A.9 Evaluation and Expected Result
The statistics that are collected by Polly (-mllvm -stats) show how often assumption were needed to apply polyhedral optimizations as well as which assumptions have been taken. To output such information per source location use the remark system of LLVM (-Rpass-analysis=polly). More sophisticated experiments are described here:

github.com/jdoerfert/CGO17_ArtifactEvaluation

A.10 Experiment Customization
The compiler can be run on other C/C++ benchmarks to evaluate the effects there.

A.11 Notes
Please see

github.com/jdoerfert/CGO17_ArtifactEvaluation
for more information, scripts and other resources.