Polyhedral Binary Decision Diagrams for Representing Non-Convex Polyhedra

Shubhang Kulkarni*
smkulka2@illinois.edu
University of Illinois, Urbana-Champaign
USA

Michael Kruse
michael.kruse@anl.gov
Argonne National Laboratory
USA

ABSTRACT

Vector sets arising in application domains such as polyhedral model optimizations are not necessarily convex, thereby ruling out representation by a single polyhedral set. In such scenarios, a commonly employed representation is using multiple polyhedral sets whose union is the vector set to be represented. While this suffices to represent all relevant vector sets for most applications, it is not necessarily efficient. Despite being simple to construct, these representations may require an exponential number of polyhedral sets to entirely cover non-convex vector sets.

In this work we introduce the *Polyhedral Binary Decision Diagram* (PBDD), inspired by binary decision diagrams (BDDs) and quasi-affine selection trees, as an alternative representation of non-convex vector sets. We implement a proof of concept in Python and show that, when supported by simple structural simplification operations from BDD literature, the PBDD avoids the exponential blow-up arising from compiling a simple program in Polly—in fact, the representation of a set's complement can be computed in constant time. Several case studies compare the scaling behavior of `isl`'s union-of-convex-polyhedra implementation and our PBDD implementation with and without simplifications.

CCS CONCEPTS

• **Software and its engineering** → *Dynamic compilers*.

KEYWORDS

scalability, polyhedral compilation, data structures, decision trees

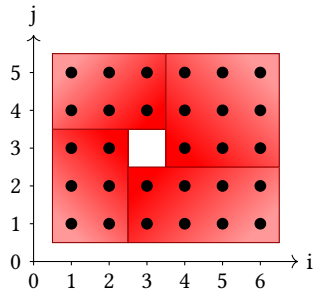
ACM Reference Format:
Shubhang Kulkarni and Michael Kruse. 2022. Polyhedral Binary Decision Diagrams for Representing Non-Convex Polyhedra. In *Proceedings of Workshop on Polyhedral Compilation Techniques (IMPACT'22).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION & MOTIVATION

Leslie Lamport may have been the first to explore the idea to represent the set of execution instances in one or more for-loops as an integer vector space bounded by constraints. Such spaces, or $\mathbb{Z}$-polyhedra, have been well explored in mathematics. Lamport used the idea to find a parallel cross section of the execution space that does not cut dependencies required for correctness [20], now called the hyperplane method. Paul Feautrier further developed the technique by using the Farkas lemma to more efficiently represent the solution space [8, 9] and used linear programming tools such as the simplex algorithm to select the best solution according to some optimization goal. This principle is referred to as the polyhedral model.

```
for (int i = 1; i <= 6; ++i)
  for (int j = 1; j <= 5; ++j)
    if (i != 3 || j != 3)
      Stmt(i,j);
```

(a) Code



(b) Statement domain

Figure 1: Non-convex (union of four convex $\mathbb{Z}$-polyhedra) execution space (*domain*) of `Stmt`

Over time, researchers have extended the kinds of loop nests that can be represented by using the polyhedral model [2]. These extensions include handling of conditional execution, non-affine constraints, potentially infinite loops, and finite integer bitwidth. Unfortunately, these also increase the complexity of the constraint system that has to be represented and eventually solved. The more complex the (in-)equation system, the longer it takes to parallelize/optimize a loop nest. One of the complexities is that the shape of the vector space is not necessarily a (convex) $\mathbb{Z}$-polyhedron anymore; an example is shown in Figure 1a.

A common internal representation of such spaces is the *union of convex polyhedra*. That is, every convex polyhedron is stored separately, and the entire vector set is assumed to be the union of several polyhedra. As a result, constraints shared by all pieces are duplicated for each polyhedron, and pieces can overlap; an element of the common set can have been "added" by multiple pieces. Applying this to the example may, for instance, result in four convex pieces shown in Figure 1b. A popular library to represent vector sets using unions of convex polyhedra for polyhedral model optimizations is the Integer Set Library (ISL) [29]).

*Work done while in the W.J. Cody Associates program at Argonne National Laboratory.

IMPACT'22, January 2022, Budapest, Hungary

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
for (int i = 0; i < n; i+=1) {
  if (i == p0)
    continue;
  if (i == p1)
    continue;
  if (i == p2)
    continue;
  ...
  Stmt(i);
}
```

Listing 1: Motivating example

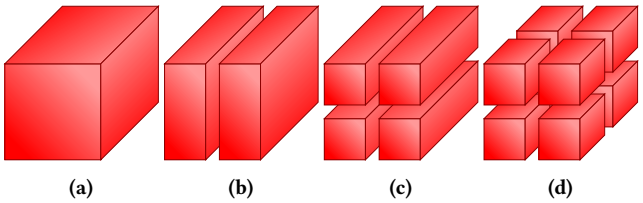


Figure 2: Iterative polygon splitting

Polly [12] is the polyhedral program optimizer for LLVM, which also uses ISL for representing its polyhedral sets. Polly itself optimizes LLVM's intermediate representation but can be enabled to automatically run when compiling with Clang along LLVM's other optimization passes.

To not be overwhelmed by overly complex programs and ensure that compilation finishes in reasonable time, Polly (like most optimization passes) employs complexity limits. If such a limit is exceeded, it stops optimizing a function. The rationale is that overly complex control flow rarely allows meaningful optimizations anyway and that constructing the polyhedral representation often results only in determining that the program cannot be optimized.

However, there are programs that we would like to optimize but that reach these complexity limits. For instance, the number of a convex polyhedron in ISL's internal representation easily grows superlinearly, as illustrated in Listing 1. The loop's iteration space is

$$\{\, \texttt{Stmt}[i] \mid 0 \leq i < n \,\},$$

but `Stmt`'s domain excludes the condition when it is skipped because of the execution of **`continue`** statements, namely, $i \neq p_k$. The variables `p0`, `p1`, and so on are defined before the loops and are unknown at compile time. Including the loop trip count n and the iteration variable itself, this spans a $p+2$-dimensional space, where p is the number or parameters.

ISL cannot directly represent unequal ($\neq$) constraints; therefore it is split into two inequalities:

$$i < p_k \;\; \texttt{OR} \;\; p_k < i. \tag{1}$$

As a union of convex polyhedra, the iteration space to accurately represent the iteration space of `Stmt` with one parameter p_0 is

$$\bigcup \begin{cases} \{\, \texttt{Stmt}[i] \mid 0 \leq i < n \;\; \texttt{AND} \;\; i < p_0 \,\} \\ \{\, \texttt{Stmt}[i] \mid 0 \leq i < n \;\; \texttt{AND} \;\; i > p_0 \,\}. \end{cases}$$

The set is illustrated in Figure 2b. With each additional equality the number of convex polyhedra grows by a factor of 2, that is, 4 convex polyhedra with $p = 2$ (Figure 2c) and 8 polyhedra in Figure 2d, namely, 2^p convex polyhedra for the general case, which is exponential growth.[1]

In cases where this occurs regularly, Polly uses two ISL objects representing a single logical set: one containing positive constraints and a second for negative conditions. The logical set they represent is the negative constraint set from the positive constraint set. Explicitly building the logical set would of course result in the same exponential explosion as in Figure 2, so it is avoided.

Eventually, an integer linear program (intLP) still needs to be solved, but a surprising majority of operations in Polly are intersection, union, complement, and subtraction. The focus of this work is to avoid the exponential growth due to these base operations before even reaching the intLP solver. We leave adapting intLP algorithms such as simplex and Fourier–Motzkin elimination to our representation as future work, although efficiency is already improved from having fewer constraints due to simplifications.

Our idea is to use a binary decision diagram (or BDD; more on BDDs in Sections 2.2 and 3) to determine whether a vector is included — in or outside the set, which we call a *polyhedral binary decision diagram* (PBDD). In a PBDD, a nonleaf (nonterminal) node represents a constraint that is evaluated to either true or false. A PBDD has exactly two leaf/terminal nodes, *IN* and *OUT*, representing the outcome of the evaluation.

In contrast to a classic BDD, node evaluations are not independent of each other. For instance, nodes representing the constraints $p_0 < i$ and $p_0 > i$ cannot be true at the same time. This fact can be used for graph simplifications in addition to those allowed by a generic BDD. For instance, $p_0 > i$ can be forwarded to its *false* branch for paths from the root through the *true* branch of $p_0 > i$.

Figure 3 shows two possible PBDDs both representing the same domain of `Stmt` in Listing 1 with 3 continue conditions, as visualized in Figure 2d. The PBDD in Figure 3a is a direct representation of the decisions for whether the statement is executed; Figure 3b shows that even when splitting equalities into inequalities as ISL does, the PBDD size can still be linear in the number of conditions by using a DAG instead of a tree. In this representation, a path represents an intersection of half-spaces only, which makes each path to the *IN* node represent a convex polyhedron from the union of convex polyhedra representation—there is an exponential number of such paths. Hence, enumerating all paths or expanding the underlying DAG into a tree should be avoided.

Union, intersection, and subtraction are simple operations that just concatenate two BDDs; the complement can be built by swapping the IN and OUT nodes. In a union of convex polyhedra representation, the individual polyhedra may overlap, whereas in a PBDD by construction each path to a terminal represents a disjoint set. This can be an advantage with some operations that require these to be disjoint, such as counting the number of $\mathbb{Z}^n$ points in the represented set [30].

Our hope is that using a PBDD internal representation makes most operations in Polly — and by extension also for polyhedral

[1] Since each constraint references a previously unused dimension, one may alternatively state the growth as a function of the number of dimensions.

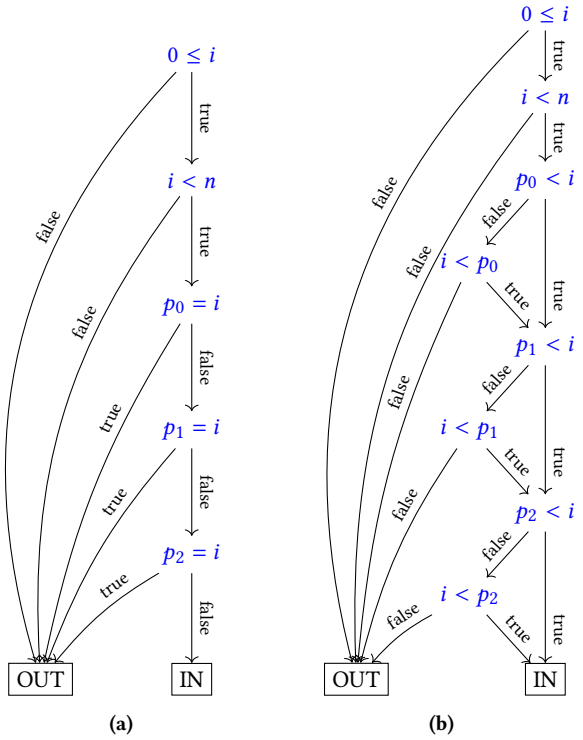


Figure 3: Possible PBDDs for Listing 1

model optimizations in general — cheaper or trivial, with little cost to other operations.

2 RELATED WORK

Our work is inspired by the *quasi-affine selection tree*[2] (QUAST) data structure [7, 25]. It is used to describe the piece-defined solution of the lexicographic minimum of a parametric $\mathbb{Z}$-polyhedron. In contrast to our PBDD, it is defined by a context-free grammar and thus always a tree instead of a DAG. A QUAST is also vector-valued whereas the PBDD is Boolean-valued representing whether a point is part of the set defined by the PBDD.

A similar data structure is the *S-tree*, or single-valued solution tree [16]. Like the QUAST, it is defined by a context-free grammar but maps to only an integer value instead. The publication [28] uses a binary *search tree* to evaluate set membership, but with the goal of reducing the number of evaluations required (i.e.,tree depth), essentially a binary space partition.

2.1 Polyhedral Optimization

Polyhedral optimizers have developed various strategies to handle non-convexity that arises from source code representations. The most common approach is to not support them at all, as done by MLIR [21] and optimizers that are based on PIPLib [25], such as PLuTo [3]. These optimizers are not able to process our motivating example in Listing 1 for any number of continue statements and have to leave it unoptimized.

[2]sometimes also quasi-affine *solution* tree [16, 31]

As already mentioned, the ISL [29] supports disjunctions in the form of union of convex polyhedra. In most cases, Polly [12] using ISL bails out when the number of polyhedra grows too large, where large means greater than a hard-coded constant such as 4 or 8. Because negative conditions are common here, however, Polly has a special treatment for the execution context that is later used to generate a runtime condition that decides whether to execute the optimized or fallback version. It has a *positive* context and a *negative* context, and the true context is a negative context set subtracted from the positive context, but never explicitly built. Hence, unequal constraints such as Equation (1) can go into the negative context as $i = p_k$ without ever needing to be split into inequalities.

2.2 Binary Decision Diagrams

Boolean functions frequently play crucial roles in several domains of computer science. However, in practical scenarios witnessing the application/computation of Boolean functions, it is often infeasible to represent these functions by classical algebraic expressions (via *switching/Boolean algebras*) or entire truth tables. Binary decision diagrams are fundamental data structures that have seen remarkable success in succinctly representing Boolean functions encountered in practice [5]. BDDs are widely used in logic synthesis [24], formal verification [14], and circuit optimization [10, 19, 23, 26].

Informally, BDDs are rooted DAGs consisting of several decision nodes and two terminal nodes labeled `true` or `false`. Each decision node is labeled by a Boolean variable and has two child nodes called the `true`-child and the `false`-child. A single directed edge from a decision node to its `true`-child (resp. `false`-child) represents an assignment of the labeled variable to `true` (resp. `false`). Given an input Boolean vector, the function evaluation is the label of the terminal obtained by traversing the path obtained as follows: Start at the root, and when at a decision node take the edge to the 0-child or 1-child as given by the value of the input on the labeled decision variable. We refer the reader to [18] for an overview of BDDs.

3 PBDD DATA STRUCTURE

Formal Definition. A PBDD data structure is defined by $Q = (V, E_0, E_1, c, r, t_{\mathsf{IN}}, t_{\mathsf{OUT}})$. Here, V denotes the node set, and $E = E_0 \cup E_1$ denotes the set of directed edges. The edges in E_1 and E_0 are the ones labeled `true`, respectively `false`. The nodes $V \setminus \{t_{\mathsf{IN}}, t_{\mathsf{OUT}}\}$ are called *constraint nodes*. Each constraint node is labeled by a constraint c_v given by the mapping function c. The resulting graph $D_Q = (V, E)$ is a directed acyclic graph, and every node except t_{IN} and t_{OUT} has exactly two outgoing edges. Furthermore, the *root* $r \in V$ denotes the unique source node, and *terminals* $t_{\mathsf{IN}}, t_{\mathsf{OUT}} \in V$ denote the two unique sink nodes (terminals) in D_Q. We denote $v[0]$ and $v[1]$ as the nodes in V obtained by traversing the *false*-labeled and *true*-labeled edges from v, respectively. We say that a PBDD $Q = (V, E_0, E_1, c, r, t_{\mathsf{IN}}, t_{\mathsf{OUT}})$ is a *representation* of vector set $P \subseteq \mathbb{Z}^n$ if for each $x \in \mathbb{Z}^n$ it holds that $x \in P$ if and only if there exists a path from r to t_{IN} in the DAG D_Q s.t. for each constraint node v on the path, if the path traverses $(v, v[1])$, then $x \in c_v$; otherwise the path traverses $(v, v[0])$ and $x \notin c_v$. We denote this characterization as $x \in P \leftrightarrow Q(x) = 1$. Note that PBDD representations are not unique, as illustrated by Figure 3.

Shannon Expansion. The *Shannon expansion* allows for using Boolean algebra within the set theoretic interpretation of PBDDs and can thus be used to argue correctness of our implementations of set theoretic as well as the structural simplification operations on PBDDs.

Consider PBDD $Q = (V, E_0, E_1, c, r, t_{\texttt{IN}}, t_{\texttt{OUT}})$. For any node $v \in V$, the *sub-DAG* obtained by removing all nodes that can reach v is a PBDD rooted at v with the same terminals $Q_v = t_{\texttt{IN}}, t_{\texttt{OUT}}$. Note that Q_v is a representation of some polyhedron $P_v \supseteq P$. The *Shannon expansion* for Q can then be expressed as follows:

$$\begin{aligned} x \in P &\leftrightarrow x \in \left(c_r \cap P_{r[1]}\right) \cup \left(\overline{c_r} \cap P_{r[0]}\right) \\ &\leftrightarrow x \in \left(c_r \cap P_{r[1]}\right) \ \texttt{OR}\ x \in \left(\overline{c_r} \cap P_{r[0]}\right) \\ &\leftrightarrow \left(x \in c_r \ \texttt{AND}\ x \in P_{r[1]}\right) \ \texttt{OR}\ \left(x \notin c_r \ \texttt{AND}\ x \in P_{r[0]}\right). \end{aligned}$$

Alternatively, we may express the above as follows:

$$Q(x) = 1 \leftrightarrow \ \texttt{OR} \begin{cases} x \in c_r \ \texttt{AND}\ Q_{r[1]}(x) = 1 \\ x \notin c_r \ \texttt{AND}\ Q_{r[0]}(x) = 1 \end{cases}.$$

Note that the Shannon expansion can be applied recursively at all nodes $v \in V$ to characterize membership in P_v according to the sub-DAG Q_v. This expansion allows us to argue correctness of set operations (Section 4.2) on PBDDs as well as perform various structural simplifications (Section 4.3) on PBDDs while preserving representation of the underlying polyhedron.

4 IMPLEMENTATION OF A PBDD

We implemented[3] our proof of concept in Python. To avoid having to implement an integer linear program solver, we fall back to IslPy for functionality that is not related to the PBDD representation. IslPy [17] is a Python binding for the ISL library. In contrast to the binding that comes with the ISL library itself, IslPy has a package in PyPI that is easy to install and exposes more functionality that the official binding does not make available.

In the following sections we discuss details of our implementation, including the type and reference structure between node objects (Section 4.1), the implementation of several set operations (Section 4.2), and simplifications to control the growth explosion (Section 4.3).

4.1 Implementation Structure

Object Types. We implemented the PBDD data structure using two types of object. The first type is `Nodes` labeled with constraints and having pointers to other `Node` objects. This `Node`-to-`Node` linking gives rise to the directed graph structure. Our implementation also includes the terminal `Node` objects within this first type of object. The second type of object is the class representing the polyhedral set data structure PBDD itself. The PBDD object points to the root and terminal `NodeS` of the data structure and exposes an external interface for applications and our experiments. Our PBDD object does not explicitly bookkeep the vertex/edge sets of the DAG formed by the linked `Node` objects. Hence, we denote PBDDs with just their root and terminal `Nodes`; algorithmic implementation details are discussed later in this section.

[3]Our implementation can be found at `https://github.com/Shubhangk/PBDD`

Reference Structure. In our implementation, objects are referenced only downward, that is, farther away from the root in the DAG. The advantages of such a referencing structure are twofold. The first is that this avoids circular references that cannot be freed by Python's reference-counting garbage collector and so have to be freed by its generational garbage collector. Furthermore, massive amounts of circular references would increase the work to be done by the generational garbage collector, which would also have to run more often. The second advantage is that a node can be referenced by multiple parents without knowledge of each other. Figure 4 illustrates adding a new constraint $p_2 < i$ to an existing set PBDD 1. Instead of copying the entire data structure for PBDD 2, the new constraint node references the existing objects from PBDD 1. This approach is valid if we know that the existing nodes are never changing (immutable). However, operations that usually would be implemented by changing existing nodes, such as rearranging/balancing nodes, do require making a copy of existing nodes (copy-on-write). This trade-off may well be worthwhile when client programs need to reuse related sets for multiple purposes, as often is the case of polyhedral optimization. Alternatively, we could introduce a reference counter to each node (or reuse Python's existing reference counter) that allows changing an object as long as its reference counter is one. Programs using the ISL library make use of the fact that copying is cheap (just increasing a reference counter).

Terminal `Node`*s.* Because of the referencing structure of our implementation, the natural choice of explicitly labeling the terminal `Nodes` with `IN` and `OUT` turns out to be expensive in execution scenarios. For example, the complement operation for a PBDD requires simply swapping the positions of the two terminals. However, since every `Node` in the data structure is immutable and can reach at least one of the two terminals, an explicit swap of the terminal nodes would require deep-copying the entire DAG structure. We overcome this requirement by having the PBDD object point to one of the terminals as `IN` and the other as `OUT`. With this arrangement, whether a path ending in a terminal node represents a value that is inside or outside the set is not determined by the terminal itself but by the PBDD object. Such a design choice enables implementing the complement operation for our PBDD by simply exchanging the pointers of the terminal nodes without having to copy any constraint node.

4.2 Implementation of Set Operations

In this section we describe the implementation of a few important set operations on PBDDs that we use in our experiments in Section 5.

Initialization. The simplest cases of PBDDs that we construct are those representing the entire `universe`, the `empty` set, or a half-space represented by a single constraint. The `init` function constructs the PBDD tuple representing each of these cases according to the input c.

$$\texttt{init}(c) := \begin{cases} \texttt{PBDD}(t_{\texttt{IN}}, t_{\texttt{IN}}, t_{\texttt{OUT}}) & c \text{ is } \texttt{universe} \\ \texttt{PBDD}(t_{\texttt{OUT}}, t_{\texttt{IN}}, t_{\texttt{OUT}}) & c \text{ is } \texttt{empty} \\ \texttt{PBDD}(\texttt{Node}(c, t_{\texttt{IN}}, t_{\texttt{OUT}}), t_{\texttt{IN}}, t_{\texttt{OUT}}) & c \text{ is a constraint} \end{cases}$$

We remark that PBDDs representing more complicated polyhedra can be constructed by first initializing each constraint using the `init` function, followed by applying set operations on the PBDD representations. For example, the PBDD representation of a convex polyhedron can be formed by repeatedly applying the `intersect` operation to the PBDD representations of its constituent inequality constraints.

Union and Intersection. We describe the `intersect` operation and remark that `union` is similar. Recall that our input is (1) PBDD $Q^1 = (r^1, t^1_{\texttt{IN}}, t^1_{\texttt{OUT}})$ representing polyhedron P^1, and (2) PBDD $Q^2 = (r^2, t^2_{\texttt{IN}}, t^2_{\texttt{OUT}})$ representing polyhedron P^2. Our goal is to output $Q = (r, t_{\texttt{IN}}, t_{\texttt{OUT}})$ representing polyhedron $P^1 \cap P^2$.

The natural algorithm is as follows: (1) redirect all arcs that come into $t^1_{\texttt{IN}}$ to r^2; next, (2) redirect all arcs that come into $t^1_{\texttt{OUT}}$ to $t^2_{\texttt{OUT}}$ and then return $Q = (r^1, t^2_{\texttt{IN}}, t^2_{\texttt{OUT}})$ as the PBDD representing the intersection. Because of the referencing structure of our PBDD implementation, however, we implement this algorithm recursively while also using copy-on-write. We describe the recursive algorithm below.

$$\texttt{intersect}\left(Q^1, Q^2\right) := PBDD\left(f_\cap(r^1), t^2_{\texttt{IN}}, t^2_{\texttt{OUT}}\right)$$

$$f_\cap(v) := \begin{cases} r^2 & v = t^1_{\texttt{IN}} \\ t^2_{\texttt{OUT}} & v = t^2_{\texttt{OUT}} \\ \texttt{Node}\left(c_v, f_\cap\left(v[1]\right), f_\cap\left(v[0]\right)\right) & \text{o.w.} \end{cases}$$

In this algorithm, the node construction function $f_\cap$ recursively builds a new graph, starting with the r^1, the root node of Q^1, and applies itself recursively. When encountering the terminal node $\texttt{IN}^1$, it is reconnected to r^2. Then, setting $t^2_{\texttt{IN}}$ to be the `IN` terminal node of the resulting PBDD ensures that to be contained in the resulting set, a root-to-terminal path must end in $t^2_{\texttt{IN}}$ Similarly, setting $t^2_{\texttt{OUT}}$ to be the `OUT` terminal node of the resulting PBDD ensures that to be excluded from the resulting set, a root-to-terminal path must end in $t^2_{\texttt{OUT}}$ Since no nodes initially in Q^2 have to be changed, they can simply be reused under the assumption of immutability. A root-to-terminal path ending in $t^1_{\texttt{OUT}}$ should result in the vector not being in the set. Thus, $t^1_{\texttt{OUT}}$ is replaced by $\texttt{OUT}^2$ so that the resulting PBDD has a single `OUT` terminal. To avoid the exponential time complexity due to overlapping recursive calls of $f_\cap$, we implement the algorithm using memoization. Thus our `intersect` implementation has runtime linear in the number of nodes in Q^1 and is independent of the size of Q^2.

Complement. As discussed previously, we implement the `complement` operation by simply exchanging the roles of the terminals in the PBDD object. Formally, this is defined as follows: On input PBDD $Q = (r, t_{\texttt{IN}}, t_{\texttt{OUT}})$,

$$\texttt{complement}(Q) := PBDD(r, t_{\texttt{OUT}}, t_{\texttt{IN}})$$

We emphasize that this definition does not violate node immutability because the terminals themselves are not changed, but their roles are swapped in the output PBDD. Thus the entire operation takes $O(1)$-time.

Emptiness Check. For the emptiness check, we make use of the following observation. Let $Q_P = (r, t_{\texttt{IN}}, t_{\texttt{OUT}})$ be a PBDD representing polyhedron P. Then, P is empty if and only if no vector can ever evaluate to be inside P; in other words, no vector can simultaneously satisfy each constraint along any r-to-$t_{\texttt{IN}}$ path. Thus, every r-to-$t_{\texttt{IN}}$ path must represent the empty set. This allows us to implement `is_empty` as follows:

$$\texttt{is\_empty}(Q) := f_\emptyset\left(r[1], c_r, \texttt{empty}\right)\ \texttt{AND}\ f_\emptyset\left(r[0], \texttt{universe}, c_r\right)$$

$$f_\emptyset\left(v, P_T, P_F\right) := \begin{cases} \texttt{true} & v = t_{\texttt{OUT}} \\ P_T \subseteq P_F & v = t_{\texttt{IN}} \\ \texttt{AND} \begin{cases} f_\emptyset\left(v[1], c_v \cap P_T, P_F\right) \\ f_\emptyset\left(v[0], P_T, c_v \cup P_F\right) \end{cases} & \text{o.w.} \end{cases}$$

In this algorithm, the subproblem $f_\emptyset(v, P_T, P_F)$ represents the r-to-v path P obtained by taking the `true` branches at the `Node`s represented by the constraints in P_T and the `false` branches at the `Node`s represented by the constraints in P_F. $f_\emptyset(v, P_T, P_F)$ evaluates to `true` if *every* r-to-$t_{\texttt{IN}}$ path having P as a subpath is empty. In particular, when at node v, the function $f_\emptyset$ recursively applies itself to all nodes below v in the PBDD. When applying itself in the `true` branch of v, $f_\emptyset$ has the constraint c_v intersected into P_T. On the other hand, when applying itself in the `false` branch of v, $f_\emptyset$ has the constraint c_v unioned into P_F. Thus when $f_\emptyset$ encounters the terminal t_{in}, it needs to check whether $P_T \cap \overline{P_F} = \emptyset$. This is equivalent to checking whether $P_T \subseteq P_F$. Note that in our PBDD implementation, P_T and P_F are polyhedral sets represented by using ISL.

The rationale for performing the inclusion check at the terminal node instead of the emptiness check is that complementing the constraints in P_F reproduces the same problem we faced in the motivating example. In particular, if the constraints in P_F are equality constraints, then the ISL representation for P_F blows up exponentially in the number of constraints. By performing the subset check on P_F, we bypass the need to explicitly construct the representation, and thus we avoid the growth explosion. Thus, the overall runtime of the algorithm is $O(|\mathcal{P}|)$, where $\mathcal{P}$ is the set of all r-to-terminal paths in the input PBDD.

Subtract. Our input is (1) PBDD Q^1 representing polyhedron P^1 and (2) PBDD Q^2 representing polyhedron P^2. Our goal is to output a PBDD representing polyhedron $P^1 \backslash P^2$. We implement this as follows:

$$\texttt{subtract}\left(Q^1, Q^2\right) := \texttt{intersect}\left(\texttt{Q}^1, \texttt{complement}\left(\texttt{Q}^2\right)\right)$$

Performing the `complement` operation takes $O(1)$ time. Thus our `subtract` implementation follows the runtime of `intersect` and has runtime linear in the number of nodes in Q^1, independent of the size of Q^2.

Subset Check. Our input is (1) PBDD Q^1 representing polyhedron P^1 and (2) PBDD Q^2 representing polyhedron P^2. Our goal is to output `true`/`false` according to whether $P^1 \subseteq P^2$. We implement this as follows:

$$\texttt{is\_subset}\left(\texttt{Q}^1, \texttt{Q}^2\right) := \texttt{is\_empty}\left(\texttt{subtract}\left(Q^1, Q^2\right)\right)$$

The `subtract` operation takes time linear in the size of Q^1 and results in a PBDD with size being the sum of the sizes of Q^1 and

Q^2 since, internally, the `intersect` operation concatenates the two PBDDs together. Thus the overall runtime is $O(|\mathcal{P}|)$, where $\mathcal{P}$ is the set of all r-to-terminal paths in the PBDD arising from `subtract`, following the analysis of `is_empty`. Note that this operation also allows us to implement the *equality checks* between two PBDDs in the same asymptotic runtime as follows:

$$\texttt{is\_equal}\left(\texttt{Q}^1, \texttt{Q}^2\right) := \texttt{is\_subset}\left(Q^1, Q^2\right) \texttt{ AND is\_subset}\left(Q^2, Q^1\right).$$

Project Out. For vector $y \in \mathbb{Z}^{n-1}$ and $x \in \mathbb{Z}$, let $y \circ x \in \mathbb{Z}^n$ denote the vector obtained by appending x to the vector y such that x becomes the final coordinate in the resulting vector. With this notation, the input to the `project_out` operation is PBDD $Q = (r, t_{\texttt{IN}}, t_{\texttt{OUT}})$ representing polyhedron $P \subseteq \mathbb{Z}^n$. The goal is to output PBDD Q' representing polyhedron $P' = \{y \in \mathbb{Z}^{n-1} : \exists x \in \mathbb{Z}\ \ y \circ x \in P\}$. Since the dimensions of the polyhedron P are ordered, our formulation makes a simplifying assumption without loss of generality. This assumption is that only the final (i.e., nth) dimension is being projected out. Note that since the dimensions of the vector space are ordered, the fact that this works for projecting out only the last dimension can be removed with appropriate reordering of the dimensions. Furthermore, this can also be iteratively reapplied to `project_out` any number of dimensions.

We have the following.

$$\begin{aligned}
P' &= \{y : \exists x \ \ y \circ x \in P\} \\
&= \left\{y : \exists x \ \ y \circ x \in \left(\left(c_r \cap P_{r[1]}\right) \cup \left(\overline{c_r} \cap P_{r[0]}\right)\right)\right\} \\
&= \left\{y : \exists x \ \ \left(y \circ x \in c_r \cap P_{r[1]}\right) \texttt{ OR } \left(y \circ x \in \overline{c_r} \cap P_{r[0]}\right)\right\} \\
&= \left\{y : \left(\exists x \ \ y \circ x \in c_r \cap P_{r[1]}\right) \texttt{ OR } \left(\exists x \ \ y \circ x \in \overline{c_r} \cap P_{r[0]}\right)\right\} \\
&= \left\{y : \left(\exists x \ \ y \circ x \in c_r \cap P_{r[1]}\right)\right\} \cup \left\{y : \left(\exists x \ \ y \circ x \in \overline{c_r} \cap P_{r[0]}\right)\right\} \\
&= \texttt{project_out}\left(\texttt{c}_\texttt{r} \cap \texttt{P}_{\texttt{r}[1]}\right) \cup \texttt{project_out}\left(\overline{\texttt{c}_\texttt{r}} \cap \texttt{P}_{\texttt{r}[0]}\right)
\end{aligned}$$

Here the first equation is by the definition of `project_out`. The second equation is by the Shannon expansion of PBDD Q. The third equation follows from distributivity of the existential quantifier over logical `OR`. The fourth equation follows by definition of union and logical `OR`. The final equality is by definition of `project_out`.

Thus, the Shannon expansion allows for breaking the operation into independent subproblems. We implement the operation recursively according to this recursive structure. When at any constraint node v of the PBDD during a recursive call, if the dimension to be projected out is not present in c_v, we simply ignore the node and recurse directly on its children. Otherwise, if the dimension is present, then we first recurse on $v[1]$ while passing in the constraint c_v and next recurse on $v[0]$ while passing in $\overline{c_v}$. On evaluation of these recursive calls, we return the `union` of the returned PBDDs

The base cases of the recursion are the terminals $t_{\texttt{IN}}, t_{\texttt{OUT}}$, which become the terminals of the resulting PBDD. If at $t_{\texttt{OUT}}$, we simply return the same terminal. Otherwise, when at $t_{\texttt{IN}}$, we `intersect` all the constraints that have been passed down through the recursive calls (since these contain the dimension to be projected out). We `project_out` the dimension from the resulting polyhedra and compute the PBDD representation with $t_{\texttt{IN}}, t_{\texttt{OUT}}$ as the `IN`-terminal and `OUT`-terminal, respectively. We return this PBDD as the evaluation of the base case. If no constraints were passed down, we simply return $t_{\texttt{IN}}$.

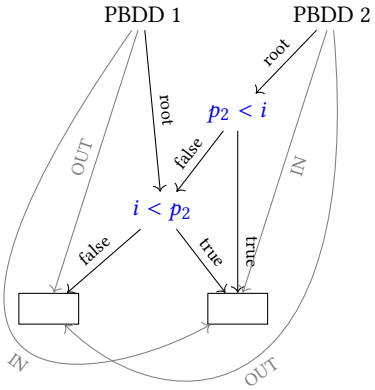


Figure 4: Data structure

Note that each path in the recursion tree of the algorithm corresponds to a path in the PBDD. Thus, the time complexity of our implementation is $O(|\mathcal{P}|)$, where $\mathcal{P}$ is the set of all paths in Q.

4.3 Simplification

Simplification is necessary to stop the PBDD from growing uncontrollably. For instance, adding tautological or redundant constraints should not add computational overhead to every following operation. While each simplification by itself may result in a PBDD with just one less constraint node, repeated application of multiple rules can dramatically reduce the number of root-to-terminal paths. In this section we describe simple pruning operations that we perform during computation to control the growth explosion of PBDDs.

Consider PBDD $Q = (r, t_{\texttt{IN}}, t_{\texttt{OUT}})$ representing polyhedron P. We say that a constraint node v of Q is *redundant* if $v[0] = v[1]$. We say that constraint nodes u, v of Q are *isomorphic* if either of the following holds:

- $c_u = c_v$; $u[1] = v[1]$; and $u[0] = v[0]$.
- $c_u = \overline{c_v}$; $u[1] = v[0]$; and $u[0] = v[1]$.

A PBDD having isomorphic or redundant nodes can be structurally simplified because of the Shannon expansion. For example, in the case of redundant node v, we have that

$$P_v = (c_v \cap P_{v[1]}) \cup (\overline{c_v} \cap P_{v[0]}) = (c_v \cap P_{v[1]}) \cup (\overline{c_v} \cap P_{v[1]}) = P_{v[1]}.$$

Here the equality conditions follow by the Shannon expansion, redundancy of v, and disjunction of a constraint with its complement being the `universe`. Thus, the node v can be pruned out of Q. The case of isomorphic nodes u, v can also be analyzed in a similar fashion to reveal that one of the nodes, say v, can be pruned and u reused in its place without changing the represented polyhedra (See Figure 5).

We now describe the `simplify` algorithm that takes as input PBDD $Q = (r, t_{\texttt{IN}}, t_{\texttt{OUT}})$ representing polyhedron P and outputs PBDD Q' with no redundant or isomorphic nodes that also represents polyhedron P.

$$\texttt{simplify}(Q) := PBDD(f(v), t_{\texttt{IN}}, t_{\texttt{OUT}})$$

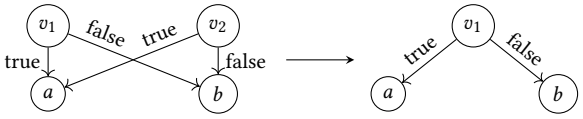


Figure 5: Constraint nodes v_1 and v_2 can be merged into a single node with constraint c_{v_1} if constraints c_{v_1} and c_{v_2} are the same.

$$f(v) := \begin{cases} v & v \text{ is terminal} \\ f(v[1]) & f(v[0]) = f(v[1]) \\ \mathtt{m}_{\cong}[c_v, f(v[1]), f(v[0])] & \mathtt{m}_{\cong}[c_v, f(v[1]), f(v[0])] \neq \mathtt{nil} \\ v & f(v[0]) = v[0], f(v[1]) = v[1] \\ \mathsf{Node}(c_v, f(v[1]), f(v[0])) & \text{otherwise} \end{cases}$$

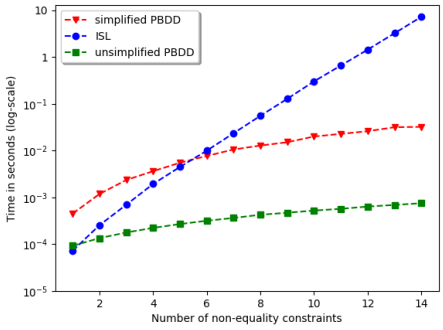
Here the function f recursively applies itself to every node, with the base case being the terminals, and prunes away redundant and isomorphic nodes. In case (1), the terminals are left unchanged by f and so remain the same for the final PBDD returned by `simplify`. Case (2) checks whether the two nodes returned by recursive calls to f are the same. If so, then v is a redundant node by definition and is ignored by directly returning the child node. Case (3) checks for isomorphic nodes. In particular, $\mathtt{m}_{\cong}$ is a lookup table that maps constraint and 2 nodes (c_u, v, w) to a node z representing that z is the canonical representative of all nodes having constraint c_u and true/false children as v and w, respectively. In case (3), if this entry is nonempty, then such a representative exists and is directly returned. Note that the $\mathtt{m}_{\cong}$ table is filled up when in cases (3), (4), and (5); in other words, if $\mathtt{m}_{\cong}(c_u, v, w) = \emptyset$, then we set $\mathtt{m}_{\cong}(c_u, v, w) := u$. Cases (4) and (5) represent scenarios when simplifications have not and have happened below node v, respectively.

Redundant Branches. In addition to local simplification operations, we implement global pruning procedures. By enumerating over paths and constructing the corresponding polyhedra along each path, we are able to prune away those paths that result in the same polyhedron. We note that ISL has a similar simplification operation whereby it attempts to prune away redundant polyhedra from its representation. We note that path enumeration may be expensive in the worst case where the number of root-to-terminal paths may be superpolynomial in the size of the PBDD. However, we experimentally observe in Section 5 that this operation, when coupled with `simplify`, helps control the number of paths and does not degrade performance too much. In fact, in a particular scenario, we find that this operation is necessary to even keep the PBDD representation computationally feasible to use (see Section 5.1).

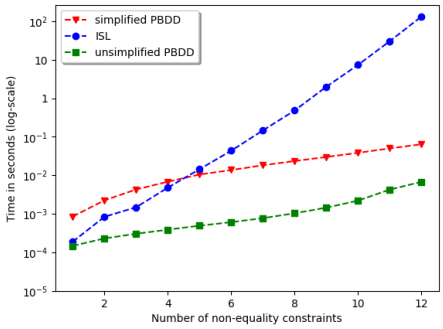
5 EXPERIMENTS

In this section we provide experimental evidence that the PBDD data structure provides tremendous speedup to execution flows that arise in practical polyhedral compilation scenarios. We provide several case studies and include performance comparisons of our Python-based PBDD implementation (with and without structural simplification operations) to the ISL (via IslPy) union-of-convex-polyhedra representation in Figures 6 to 8.

In each figure the plots depict the trade-off between the number of constraints in the polyhedra (x-axis) vs. the time taken (y-axis, $\log_{10}$ of the number of seconds) to run the specific execution flow described in the experiment. The blue, red, and green dashed lines indicate the performance of ISL, our PBDD implementation with and without simplifications, respectively.



(a) Iterative Intersection of Complement (Algorithm 5.1)



(b) Iterative Subtraction of Redundant Constraints (Algorithm 5.2)

Figure 6: Vector set construction

While the test cases in this section are synthetic in the sense that they are designed to reveal worst-case scenarios, these are also condensed from sets that occur when running Polly, in particular when optimizing the motivating example Listing 1. We also traced the sequence of function calls to ISL during the execution of Polly using library interposition. This results in a trace file that can be compiled and executed to repeat the same polyhedral operations as Polly did. We converted parts of it to use our PBDD implementation instead to see how it behaves in practice. It turned out that the simplification operations described in Section 4.3 are crucial to maintaining compact representations of the underlying polyhedra.

5.1 PBDD Construction

We focus on the motivating example from Section 1 (Listing 1) where the underlying iteration domain was a non-convex polyhedron in the form of a conjunction of several nonequality constraints. The represented vector set should look like that in Figure 2 and its PBDD one of Figure 3.

In the first experiment we construct this set by consecutively intersecting the complement of the equality $c_i \mapsto (p_i = i)$, namely, Equation (1):

Algorithm 5.1 (Iterative Intersection of Complement)

(1) Input: k, constraints c_i
(2) $P :=$ `Universe`
(3) for $i = 1 \dots k$
  (a) $P =$ `intersect` $(P,$ `complement`$(c_i))$
(4) Output P

In a variant construction procedure, we directly use the subtract operation (in the PBDD this is implemented by intersecting with the complement), but we first precondition the set by intersecting with the previous result:

Algorithm 5.2 (Iterative Subtraction with Redundant Constraints)

(1) Input: k, constraints c_i
(2) $P :=$ `Universe`
(3) for $i = 1 \dots k$:
  (a) $P =$ `subtract` $(P,$ `intersect`$(c_i, P))$
(4) Output P

The idea is that the additional intersection will add additional constraints to the subtraction arguments that will be redundant in the result. An effective simplification will be able to simplify these constraints such that they appear only once. The experimental results are shown in Figure 6. For Figure 6a, ISL exhibits an exponential behavior while the PBDD implementation stays subexponential (and even polynomial) in the number of nonequality constraints. There are two reasons for this behavior. First, examination of the constructed representations reveals that the size of ISL's union of convex polyhedra representation is already exponential in the number of nonequality constraints, while the PBDD representations, both simplified and unsimplified, are linear in size. Second, the intersection operation for ISL is quadratic time while that of the PBDD is linear time without simplifications (See also Section 5.4). In this scenario, the PBDD representation does not require simplifications at any steps.

In Figure 6b, one can observe that the PBDD with simplifications performs similarly to its performance in Figure 6a with only a slight degradation. This degradation happens mainly due to the fact that several redundant nodes are added into the PBDD at each iteration. These are pruned out during the simplification step carried out at the end of each `intersect`. Despite the pruning, the PBDD performs faster than ISL which takes exponential time on Algorithm 5.2. Furthermore, although the PBDD without simplifications has the best performance, one can prove by induction on the number of iterations in Algorithm 5.2 that the unsimplified PBDD representation blows up exponentially. Due to this, using the representation to perform set operations (e.g. checking `is_empty` after running Algorithm 5.2) is computationally infeasible and the simplified PBDD performs best overall. We discuss such tradeoffs further in Section 5.3.

5.2 Set-Algebra Operations (Non-Convex)

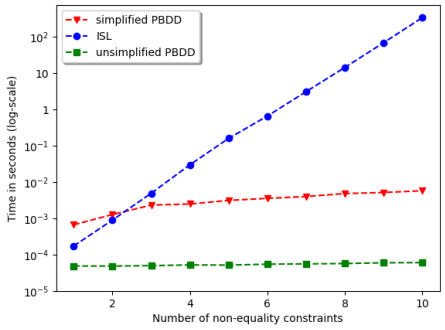
The experiments in Section 5.1 show that it is exponentially faster to construct the PBDD representation of the motivating example than the union-of-convex-polyhedra representation. The natural question that arises is whether this representation can be practically applied in computations involving set operations. In this section we take a step toward an affirmative answer to this question by focusing on set operations that frequently arise in polyhedral compilation—the domain from which this example originates. In particular, we compare the performance of these operations on the representations of the motivating example constructed in the preceding section. Besides *non-convex projection* and *emptiness check* the performances in this section do *not* include the time for construction of the representation. Figure 7 showcases the comparisons. We describe the specific experiments next.

Non-convex Intersection. We `intersect` the representations of two polyhedra constructed by Algorithm 5.1. Both polyhedra have the same number of nonequality constraints, and all the nonequality constraints are distinct. This ensures that the two polyhedra have a nonempty intersection. The plots in Figure 7a show the performance comparison of this `intersect` operation. ISL's `intersect` operation scales the worst case quadratically in the size of its representation (see Section 5.4). The experiments in Section 5.1 show that this representation has exponential size, resulting in the overall exponential behavior observed in Figure Figure 7a. Note that since all the constraints in the representation are distinct/independent, the resulting PBDD has no redundancies that need to be pruned out. Thus the simplification steps result in an additive linear overhead that leads to the simplified and non-simplified PBDD performing as observed.

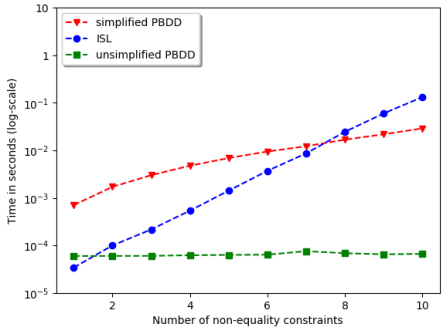
Non-convex Union. We `union` two polyhedra constructed by Algorithm 5.1 similar to *non-convex intersection* described previously. The plots in Figure 7b show the performance comparison of this `union` operation. We observe that the scaling behavior of the `union` operation on PBDD is the same as that of the `intersect` operation, leading to performances similar to those in *non-convex intersection.* However, the intersect operation for ISL is more expensive than the union operation, as can be observed by comparing both the blue curves in Figures 7a and 7b. Nevertheless, both the simplified and unsimplified PBDD implementations outperform ISL even in the union operation.

Non-convex Complement. We `complement` a polyhedron constructed by Algorithm 5.1. The plots in Figure 7c show the performance comparison. We note that the scaling behavior of the PBDD implementation is independent of the size of the representation and is constant time, reflecting the discussion of the `complement` operation in Section 4. On the other hand, ISL requires exponential time to perform this operation even though the final union-of-convex-polyhedra representation is very small.

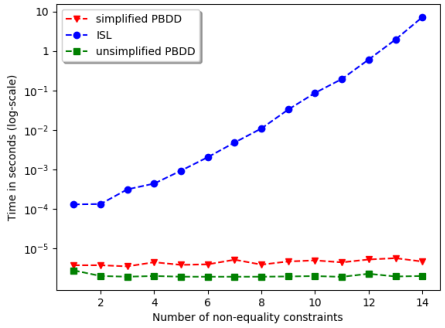
Non-convex Subtraction. We `subtract` from the `universe` a non-convex polyhedron formed by the `union` of several equality constraints. Note that this results in the same output as that of Algorithm 5.1 and Algorithm 5.2. The plots in Figure 7d show the performance comparison of this `subtract` operation. We observe that the scaling behavior of the `subtract` operation on PBDD is similar to that of the `intersect` operation, leading to performances similar to those in *non-convex intersect.* This behavior follows the implementation of `subtract` described in Section 4 along with insights from *non-convex complement.*



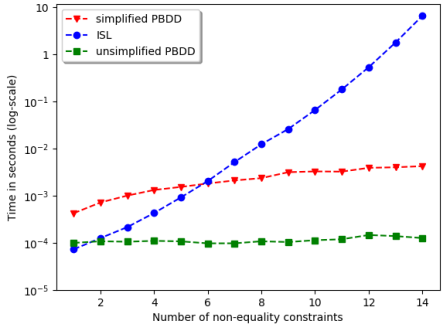
(a) Non-convex Intersection



(b) Non-convex Union

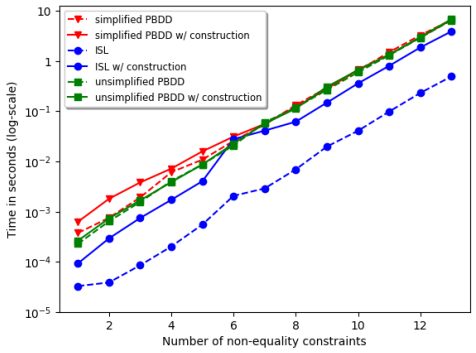


(c) Non-convex Complement

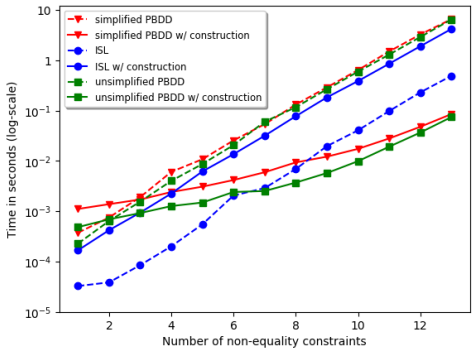


(d) Non-convex Subtraction

Figure 7: Set-algebraic operations on non-convex vector sets



(a) Non-convex project-out



(b) Non-convex emptiness check

Figure 8: Expensive set operations on non-convex sets

5.3 Projection and Emptiness

The following *non-convex projection* and *non-convex emptiness check* highlight the trade-off we currently face between representational simplicity and usability of the PBDD. In both scenarios we observe that when the performances of *only* the respective operations are measured *independently of the construction procedures*, then our current PBDD implementation performs slower than ISL. However, taking into account the time of construction—something every such polyhedral representation will witness—we observe that the PBDD performs comparable to or even better than ISL. These scenarios provide intriguing directions for future work (see Section 7).

Non-convex Projection. We `project_out` the first dimension from the representation of a polyhedron constructed by Algorithm 5.1. Figure 8a compares the performances of ISL and the PBDD implementation in two manners: (1) time to construct and `project_out`; and (2) time to just `project_out`.

Non-convex Emptiness Check. We check whether a polyhedron constructed by first running Algorithm 5.1 and then intersecting with one of the input equality constraints (resulting in the empty set) is `is_empty`. Figure 8b compares the performances of ISL and the PBDD implementation in two manners: (1) time to construct and check `is_empty`; and (2) time to just check `is_empty`.

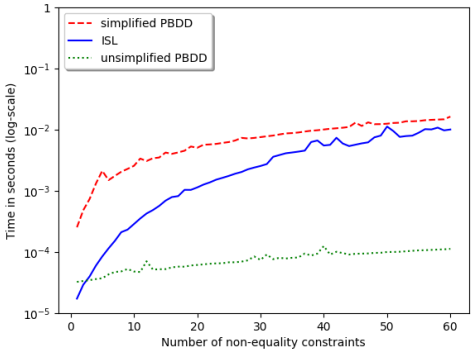


Figure 9: Convex intersection

5.4 Set-Algebra Operations (Convex)

Even with only convex polyhedra and a single vector dimension, ISL's intersect operation has quadratic complexity. To illustrate this behavior, we use the following algorithm.

Algorithm 5.3 (Convex Intersection)

(1) Input: k
(2) $P := \cup_{j=1}^{k} \{ x \mid x = a_j \}$, where a_j random from $[0, 10000]$
(3) $Q := \cup_{j=1}^{k} \{ x \mid x = b_j \}$, where b_j random from $[0, 10000]$
(4) $R =$ `intersect`(P, Q)
(5) Output R

The rationale behind the selection of such polyhedra is that it is unlikely that ISL or our PBDD implementation can find smaller representations than storing all k constraints. For instance, if instead of random numbers we chose the even numbers from 2 to $2k$, the representation as $\{ x \mid \exists y : 2y = x \}$ would be possible.

We measure only the `intersect` with the construction time for P and Q excluded, shown in Figure 9. P and Q have essentially the same representational complexity (linear in the number of constraints) in both data structures. The unsimplified PBDD has linear performance based solely on the time of `intersect`. The simplified PBDD performs similar to ISL as the simplification operations are still internally carried out to determine whether simplification is possible.

6 CONCLUSION

We demonstrated that a PBDD is a viable alternative to the union of convex polyhedra representation. Its advantages are found primarily when applying set operations of logical vector sets before eventually solving a linear equations system. Decision trees such as QUASTs and especially BDDs are well explored in the literature. The PBDD also provides a more efficient representation of non-equalities that otherwise have to be represented by a pair of inequalities that, when combined, can easily require exponentially growing costs. In the extreme case, the complexity of representing the complement of a set is reduced from exponential to constant, by swapping the role of the terminal nodes of the decision diagram.

Unfortunately, we cannot compare the total time taken for the end-to-end compilation and optimization of a program such as the one in Listing 1 using Polly as this would require us to implement and replace the entire ISL API used by Polly. In any case, with sufficiently large number of constraints the exponentially-growing union-of-convex-polyhedra representation would have dominated the total compilation time without necessarily impacting the intLP solver time, as Polly's already present split into positive and negative contexts shows (see Section 2.1).

Set-oriented operations are widely used in optimizing compilers using the polyhedral model, where the exponential complexity cases can be observed in real-world code. A PBDD representation might allow optimizing more code instead of bailing out because the compilation would take too long to complete. While our prototype is not practical as a replacement of the ISL library because of the overhead and limitations that come with Python, a more complete and efficient implementation might.

7 FUTURE WORK

So far we implemented only intersection, union, subtraction, complement, and some graph simplifications independently from ISL. Other operations, such as the emptiness check that requires a linear program solver, currently depend on the ISL implementation. ISL basic sets are constructed when necessary, and the PBDD can be reconstructed after non-projection operations. As shown, doing this piece-wise already results in significant speedups but involves additional overhead in first constructing the ISL data structure as an intermediate representation, whereas we could go directly to a simplex or PIP-style tableau implementation.

Parallelism. The recursive nature of the PBDD invites the use of divide-and-conquer style parallelism or simplifications running in the background. Unfortunately, such parallelism is blocked by Python's inability to handle in-process parallelism. However, Python's global interpreter lock might be removed in the near future [11]. Alternatively, an implementation in C/C++ would remove the interpreter overhead.

Operation Efficiency. Some operations can be constructed out of the already-implemented set operations (such as subtraction) or can be applied to the Shannon expansion recursively while memoizing multiple parents (e.g., adding a new unknown dimension without constraints); others could make better use of the recursive structure. Moreover, some algorithms may have multiple, equivalent implementations. For instance, intersection and union can have either tree appended to the other, but only one needs to be copied.

Typed Roots. Other operations can be always applied to the root node. For instance, the division of unknowns into tuples, parameter, set, and map-domain dimensions can be applied as a postoperation, while the main PBDD is untyped and its dimensions are unordered. This simplifies the implementation of the PBDD itself and allows reusing nodes for sets that use the same constraints but vary in the logical meaning of dimensions.

Simplifications. There is a trade-off between investing computation time in simplifications and suboptimal graph size reduction. Some simplifications are more costly than what they save in following operations; but, as shown in Section 5, omitting them entirely results in exponential computational dept. It may make sense to choose the one with fewer nodes or the one that is not referenced

from somewhere else so it can be modified in place. Some investigation is needed to figure out under which circumstances to use an algorithm or whether a graph simplification is useful.

In addition to the simplification operations described in Section 4.3, we implemented a procedure that checks whether entire sub-DAGs represent the same polyhedra. Specifically, we compared their explicit constructions. While this operation dramatically reduces the size of the PBDD, our relatively straightforward implementation turns out to be expensive and degrades performance beyond that of ISL. An efficient implementation of this or a similar operation could greatly reduce the sizes of PBDDs and speed up practical execution flows.

Decision Orderings. In our current implementation the structure of the PBDD is the result of the order of operations use to create them. This has the consequence that logically equivalent sets can very different internal structures with different performance characteristics. Bryant [4] showed that the BDD of a Boolean function obtained by fixing a variable ordering, often referred to as ROBDDs [5, 18], and maximally applying two simplification rules is unique. While this still would not mean that the same set would always have the same PBDD due to decision equivalences, controlling the node order potentially allows finding the cheapest PBDD representation. Unfortunately, finding the decision ordering that yields the smallest BDD has been shown to be NP-complete [6, 22, 27], but heuristics [1, 13, 15] still allow for smaller representations and thus cheaper operations, even if in worst-case their size remains exponential.

Approximations. We would like to support three kinds of approximations. One possibility is to add an additional terminal node UNDEFINED. When simplifying, nodes of this kind can combined with either an IN or OUT to simplify the PBDD. That is, if a node has edges to an IN and UNDEFINED node, the entire node can be replaced with the IN node, allowing the elimination of any constraints. Since UNDEFINED can always be combined with something else (unless the entire set itself is undefined), the information that was originally undefined can easily get lost. Therefore, an alternative is to have a second PBDD part of the root node that stores the defined subset and is used implicitly by the graph simplification. This is approximately equivalent to ISL's `gist` operation. It is useful to define the universe of possible values; for instance, the program in Listing 1 may have undefined behavior if `n <= 1` or it is known that this will never occur (e.g., the function exits early for trivial cases), which can be used to simplify the polyhedral set by removing corner cases that are known to never occur. Finally, it would be of practical interest to design operations that are allowed to return over- or underapproximations if computing the exact result would be too computationally expensive. A general example of this for vector sets could be the operations such as the emptiness check returning "maybe" results. In polyhedral model optimization, the set of dependencies can be arbitrarily overapproximated, which decreases the probability that an optimization can be applied but will never result in a miscompilation. This is in contrast to ISL's approach to return an error value when an application-defined number of elemental operations is exceeded, in which case the application does not even get an approximation. However, how hard we should search for a good approximation is application-specific, and we will need to find a measured different from ISL's number of elemental operations.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular its subproject PROTEAS-TUNE.

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. 2001. MINCE: A Static Global Variable-Ordering for SAT and BDD.

[2] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model is More Widely Applicable Than You Think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction* (Paphos, Cyprus) *(CC'10/ETAPS'10)*. Springer-Verlag, Berlin, Heidelberg, 283–303. https://doi.org/10.1007/978-3-642-11970-5_16

[3] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*.

[4] Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[5] Randal Bryant. 2003. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *Comput. Surveys* 24 (03 2003). https://doi.org/10.1145/136035.136043

[6] Michael Carbin. 2006. Learning Effective BDD Variable Orders for BDD-Based Program Analysis.

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

[8] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem – Part I: One-dimensional time. 21, 6 (Oct. 1992), 313–347. https://doi.org/10.1007/BF01407835

[9] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem – Part II: Multidimensional time. 21, 6 (Dec. 1992), 389–420. https://doi.org/10.1007/BF01379404

[10] G. Fey, Junhao Shi, and R. Drechsler. 2004. BDD circuit optimization for path delay fault testability. In *Euromicro Symposium on Digital System Design, 2004. DSD 2004.* 168–172. https://doi.org/10.1109/DSD.2004.1333273

[11] Sam Gross. [n.d.]. Python multithreading without the GIL. python-dev@python.org mailing list post. https://lwn.net/ml/python-dev/CAGr09bSrMNyVNLTvFq-h6t38kTxqTXfgxJYApmbEWnT71L74-g@mail.gmail.com/

[12] Tobias Grosser. 2011. Enabling Polyhedral Optimizations in LLVM.

[13] Orna Grumberg, Shlomi Livne, and Shaul Markovitch. 2003. Learning to Order BDD Variables in Verification. *J. Artif. Int. Res.* 18, 1 (Jan. 2003), 83––116.

[14] A.J. Hu. 1997. Formal hardware verification with BDDs: an introduction. In *1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, PACRIM. 10 Years Networking the Pacific Rim, 1987-1997*, Vol. 2. 677–682 vol.2.

[15] Peter Kissmann and Jörg Hoffmann. 2014. BDD Ordering Heuristics for Classical Planning. *J. Artif. Int. Res.* 51, 1 (Sept. 2014), 779––804.

[16] Arkady Klimov. 2015. Yet Another Way of Building Exact Polyhedral Model for Weakly Dynamic Affine Programs. *arXiv preprint arXiv:1501.03839* (2015).

[17] Andreas Kloeckner. [n.d.]. *islpy documentation.* https://documen.tician.de/islpy/

[18] Donald E. Knuth. 2011. *The Art of Computer Programming: Combinatorial Algorithms, Part 1* (1st ed.). Addison-Wesley Professional.

[19] Alexander I. Kornilov and Tatiana Yu Isaeva. 1995. *Circuit depth optimization by BDD based function decomposition.* Springer US, Boston, MA, 64–69. https://doi.org/10.1007/978-0-387-34920-6_6

[20] Leslie Lamport. 1974. The Parallel Execution of DO Loops. *Commun. ACM* 17, 2 (Feb. 1974), 83–93. https://doi.org/10.1145/360827.360844

[21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[22] H. Li and D. Probst. 1988. Abstract pecification of Synchronous Data Types for VLSI and Proving the Correctness of Systolic Network Implementations. *IEEE Trans. Comput.* 39, 06 (jun 1988), 710–720. https://doi.org/10.1109/12.2209

[23] P. Lindgren, M. Kerttu, M. Thornton, and R. Drechsler. 2001. Low power optimization technique for BDD mapped circuits. In *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*. 615–621. https://doi.org/10.1109/ASPDAC.2001.913377

[24] Shin-ichi Minato. 1993. Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams. *IEICE transactions on fundamentals of electronics, communications and computer sciences* E76, A6 (jun 1993), 967–973. http://hdl.handle.net/2115/47468

[25] Cèdric Bastoul Paul Feautrier, Jean-Fraŋ cois Collard. [n.d.]. *PIP/PipLib – A Solver for Parametric Integer Programming Problems*.

[26] Sanjeet Kumar Sinha and Suman Lata Tripathi. 2018. BDD Based Logic Synthesis and Optimization for Low Power Comparator Circuit. In *2018 International Conference on Intelligent Circuits and Systems (ICICS)*. 37–41. https://doi.org/10.1109/ICICS.2018.00020

[27] Seiichiro Tani, Kiyoharu Hamaguchi, and Shuzo Yajima. 1993. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *Algorithms and Computation*, K. W. Ng, P. Raghavan, N. V. Balasubramanian, and F. Y. L. Chin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–398.

[28] P. Tøndel, T.A. Johansen, and A. Bemporad. 2003. Evaluation of piecewise affine control via binary search tree. *Automatica* 39, 5 (2003), 945–950. https://doi.org/10.1016/S0005-1098(02)00308-4

[29] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 299–302.

[30] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica* 48, 1 (2007), 37–66.

[31] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. 2013. Array dataflow analysis for polyhedral X10 programs. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 23–34.