FPL: Fast Presburger Arithmetic through Transprecision

ARJUN PITCHANATHAN, IIIT Hyderabad, India
CHRISTIAN ULMANN, ETH Zurich, Switzerland
MICHEL WEBER, ETH Zurich, Switzerland
TORSTEN HOEFLER, ETH Zurich, Switzerland
TOBIAS GROSSER, University of Edinburgh, United Kingdom

Presburger arithmetic provides the mathematical core for the polyhedral compilation techniques that drive analytical cache models, loop optimization for ML and HPC, formal verification, and even hardware design. Polyhedral compilation is widely regarded as being slow due to the potentially high computational cost of the underlying Presburger libraries. Researchers typically use these libraries as powerful black-box tools, but the perceived internal complexity of these libraries, caused by the use of C as the implementation language and a focus on end-user-facing documentation, holds back broader performance-optimization efforts. With FPL, we introduce a new library for Presburger arithmetic built from the ground up in modern C++. We carefully document its internal algorithmic foundations, use lightweight C++ data structures to minimize memory management costs, and deploy transprecision computing across the entire library to effectively exploit machine integers and vector instructions. On a newly-developed comprehensive benchmark suite for Presburger arithmetic, we show a 5.4x speedup in total runtime over the state-of-the-art library isl in its default configuration and 3.6x over a variant of isl optimized with element-wise transprecision computing. We expect that the availability of a well-documented and fast Presburger library will accelerate the adoption of polyhedral compilation techniques in production compilers.

CCS Concepts: • **General and reference** → **Performance**; *Experimentation*; • **Mathematics of computing** → **Mathematical software performance**; **Solvers**; • **Computing methodologies** → *Vector / streaming algorithms*; *Optimization algorithms.*

Additional Key Words and Phrases: Presburger Arithmetic, Integer Sets, Transprecision, Polyhedral Compilation

ACM Reference Format:
Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser. 2021. FPL: Fast Presburger Arithmetic through Transprecision. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 162 (October 2021), 26 pages. https://doi.org/10.1145/3485539

1 INTRODUCTION

Polyhedral compilation [Verdoolaege 2016] based on Presburger arithmetic [Haase 2018] is widely used for performance optimization in high-performance computing and machine learning [Baghdadi et al. 2019; Chen et al. 2018; Grosser and Hoefler 2016; Vasilache et al. 2018], formal verification [Namjoshi and Singhania 2016], cache modeling [Gysi et al. 2019], the derivation of data movement bounds [Olivry et al. 2020], and configurable computing [Pouchet et al. 2013]. The

Authors' addresses: Arjun Pitchanathan, IIIT Hyderabad, India, arjun.p@research.iiit.ac.in; Christian Ulmann, ETH Zurich, Switzerland, christian.ulmann@inf.ethz.ch; Michel Weber, ETH Zurich, Switzerland, michel.weber@inf.ethz.ch; Torsten Hoefler, ETH Zurich, Switzerland, torsten.hoefler@inf.ethz.ch; Tobias Grosser, University of Edinburgh, United Kingdom, tobias.grosser@ed.ac.uk.

2475-1421/2021/10-ART162
https://doi.org/10.1145/3485539

performance of such techniques is important in scenarios such as just-in-time compilation, but the performance aspect has not often been a focus in the past [Lattner et al. 2021]. A large fraction of the cost of polyhedral approaches lies in computing the results of Presburger set operations. Grosser et al. [2020] ran the Polly polyhedral loop optimizer [Grosser et al. 2012] on computational kernels in Polybench [Pouchet 2012] and found that 71% of Polly's runtime was spent in the underlying math library for Presburger arithmetic. Polyhedral analyses, such as cache modeling, consist almost entirely of sequences of Presburger arithmetic operations.

We investigate how polyhedral compilers spend time in Presburger set operations by developing a comprehensive benchmark suite. We run Polly, the PPCG GPU compiler [Verdoolaege et al. 2013], and Pluto [Bondhugula and Ramanujam 2007] on Polybench [Pouchet 2012] and extract the Presburger set operations they perform. We thereby obtain a benchmark suite consisting of 1,129,239 test cases distributed across eight key operations: union, intersect, subtract, complement, equality checking, emptiness checking, eliminating existential quantifiers, and coalescing [Verdoolaege 2015]. We find that the dimensionalities of the sets involved are typically quite small, with the median set having only eight dimensions (Figure 6a). We also find that 99% of constraint coefficients in the test cases fit within just 9 bits (Figure 6d). We exploit these observations when designing and optimizing our Presburger library.

Existing Presburger libraries such as isl [Verdoolaege 2010] typically use arbitrary-precision arithmetic for all their computations since this may be necessary for correctness in the worst case. However, this is far from the common case: FPL can compute 90% of the test cases in our benchmark suite using only 16-bit machine integers, and isl spends over 74% of its runtime on such test cases (Figure 13). The performance-critical parts of a Presburger library largely consist of row operations on matrices of integers (Section 2). Using machine integers allows us to vectorize these performance-critical row operations, magnifying the performance potential of transprecision computing.

We use 16-bit integers for all our computations whenever possible and fall back to wider integer types or arbitrary-precision only if an overflow occurs. This gives us the best of both worlds: vectorized computations using machine integers for the common case, and a fallback to arbitrary-precision integers to preserve correctness in the rare case where this is needed. Grosser et al. [2020] implemented such a *transprecision* approach at two possible levels of granularity: at the matrix level, and at the element level. Both of these approaches were embedded within isl. Their matrix-level transprecision implementation focused on only one particular operation and depended heavily on C++ templates, while isl is a C library. isl implements its own hand-rolled templates using C macros, but these are difficult to understand and maintain; it would be impractical to scale this to the whole library. Their element-level transprecision approach allowed each individual integer to switch between 32-bit integers and arbitrary-precision arithmetic, which inhibits vectorization. This approach also pays a higher overhead, since every operation on integers must check the state of the transprecision integer and dispatch to the appropriate version of the operation. As these approaches were embedded in isl, they inherit its disadvantages: isl's implementation does a lot of dynamic memory management, manual reference counting, and pointer chasing.

Ideally, we would like a separate vectorized library that computes with 16-bit integers and returns an error if an overflow occurs. In case of overflow, we would then retry the same computation with a higher precision library. In this design, the overhead of transprecision is minimal as we only have to perform a single dispatch as to *which library to use*, rather than dispatching for each low-level integer operation. In the common case, this design has almost no overhead for supporting transprecision.

We introduce FPL, a Fast Presburger Library that implements transprecision computing at the *library level* and uses data structures that minimize dynamic memory allocation. In particular,

we leverage LLVM [Lattner and Adve 2004] data structures like `SmallVector`, which stores its elements inline up to a specified size and only dynamically allocates memory if the inline memory proves insufficient. Large parts of FPL have been upstreamed to LLVM's MLIR project, where it is already being used to enable affine loop fusion.

FPL implements transprecision computing by leveraging C++ templates to support instantiating the library with any integer type, and therefore providing a separate code path for each integer type. Our library also provides an API that transparently switches between integer types as needed by instantiating the library with the appropriate integer type. Overall, we find that FPL achieves a 5.4x speedup in terms of total runtime over isl with arbitrary precision arithmetic (GMP) and 3.6x over isl enhanced with the element-wise transprecision approach.

Our library follows the algorithmic design of earlier polyhedral libraries such as Omega [Kelly et al. 1996], VPL [Fouilhe 2015], Polylib [Loechner 1999] and, in particular, isl [Verdoolaege 2010]. In this work, we document the underlying algorithmic foundations and provide a performance analysis of these algorithms (Section 4), which we expect to encourage further work into optimizing Presburger libraries. Our contributions are:

- A fast transprecision Presburger library implemented in modern C++ that:
  - specializes for small values using transprecision computing at the library level, and
  - uses a lightweight design that avoids pointer indirections and leverages data structures that use fewer dynamic allocations.
- A detailed documentation and performance analysis of the algorithmic foundations of a modern Presburger library (Section 4).
- A benchmark suite for Presburger arithmetic that characterizes typical workloads in polyhedral compilation.[1]
- A detailed performance evaluation of our new library that shows a 5.4x speedup in total runtime over isl in its default configuration and a 3.6x speedup over isl enhanced with an element-wise transprecision optimization (Section 6).

2 PRESBURGER ARITHMETIC IN FPL

We describe Presburger arithmetic as implemented in FPL, laying out the kinds of constraints and sets that are allowed. We also briefly describe some key implementation specifics and outline the performance characteristics of our system. Building on this, we describe the architecture of our library (Section 3) and its algorithmic foundations (Section 4).

2.1 Basic Sets

A basic set is the set of solutions to a list of affine constraints over n integer-valued variables. Suppose the variables are $x_1, x_2, \ldots, x_n \in \mathbb{Z}$. Affine constraints can be inequalities like $a_1x_1 + \ldots + a_nx_n + c \geq 0$ or equalities such as $a_1x_1 + \ldots + a_nx_n + c = 0$, where the coefficients $a_1, \ldots a_n$ and the constant term c must be integers. Such a set corresponds to the set of integer points lying in a convex polytope. For example, consider the set $\{(x, y) \in \mathbb{Z}^2 : 1 \leq x \leq 7 \wedge x = 2y\}$, which we write for brevity as $\{(x, y) : 1 \leq x \leq 7 \wedge x = 2y\}$. This set contains the points $(2, 1)$, $(4, 2)$, and $(6, 3)$. We will refer to x and y as ordinary variables, in contrast to symbolic, existential, and division variables, as detailed below. FPL stores a basic set as two vectors of constraints: one for inequalities and one for equalities. Each constraint is stored as a vector of coefficients (integers).

Symbolic Variables. Symbolic variables correspond to fixed, but unknown values. Mathematically, a basic set with symbolic variables is like a family of basic sets indexed by the symbolic variables. For example, $(p, q) \rightarrow \{x : p \leq x \leq q\}$ is a family of Presburger sets indexed by two integers

[1] Available at https://github.com/Superty/presburger-benchmarks.

p and q, i.e., for any $p, q \in \mathbb{Z}$, $S_{p,q} = \{x : p \leq x \leq q\}$. For example, $S_{1,0}$ is the empty set and $S_{-3,-1} = \{-3, -2, -1\}$. The basic set must still be described as a conjunction of affine constraints over the ordinary and symbolic variables. A non-example is $p \rightarrow \{x : x = p^2\}$; this is not a valid basic set since the constraint is quadratic in the variable p.

Existential Quantification. Variables can be existentially quantified. For example, consider the set $\{x : \exists\, y, 1 \leq x \leq 7 \wedge x = 2y\}$. An assignment to the symbolic and ordinary variables is valid if there exists some assignment to the existentially quantified variables satisfying the constraints, so this set is equivalent to $\{2, 4, 6\}$. This set can be viewed as the result of projecting out the variable y from the set $\{(x, y) : 1 \leq x \leq 7 \wedge x = 2y\}$. In general, a set with existential variables can be viewed as the result of projecting out some variables from a basic set without existentially quantified variables.

Division Variables. We can support floor divisions of affine expressions by constants by exploiting existential quantification. As a result, we can support modulo constraints by representing them using floor divisions. For example, consider the set $\{x : x \equiv 1 \bmod 3 \wedge 2 \leq x \leq 6\}$, equivalent to $\{2, 5\}$. $\lfloor x/3 \rfloor$ is the quotient on dividing x by 3 and $x - 3\lfloor x/3 \rfloor$ is the remainder, so we can represent this set as $\{x : x - 3\lfloor x/3 \rfloor = 1 \wedge 2 \leq x \leq 6\}$. The remainder must lie between 0 and 2, so we know that $0 \leq x - 3\lfloor x/3 \rfloor \leq 2$. In fact, the only integer value of q satisfying $0 \leq x - 3q \leq 2$ is $q = \lfloor x/3 \rfloor$. Since increasing or decreasing q by one decreases or increases $x - 3q$ by three, there is exactly one value of q such that $x - 3q$ lies between 0 and 2. The above set can therefore be rewritten as $\{x : \exists\, q, 0 \leq x - 3q \leq 2 \wedge x - 3q = 1 \wedge 2 \leq x \leq 6\}$. Note that one cannot eliminate the extra variable here; the constraint that q is an integer is crucial. In general, we can support any floor divisions where the numerator is an affine expression $a_1x_1 + \ldots + a_nx_n + c$ and the denominator is a fixed positive integer d by adding an existentially quantified variable q along with the additional constraint that $0 \leq a_1x_1 + \ldots + a_nx_n + c - qd \leq d - 1$.

We refer to the variable q introduced above as a division variable, and we use the term "existential variable" only for existentially quantified variables that are not division variables. Although divisions are implemented in Presburger arithmetic as an added existentially quantified variable along with two inequalities, we distinguish between existential variables and division variables. Our implementation explicitly stores the representation of the added variable as the floor division of an affine expression by an integer. This additional information helps us handle division variables better than existential variables, and we sometimes need to convert a representation of a set using existential variables into one that only has division variables, which we refer to as eliminating existential variables Section 4.1.3. The two added inequalities are not stored explicitly, but can be recovered from the floor division representation whenever necessary. It turns out to be useful to deal with these division constraints separately from the other inequalities when computing set differences (Section 4.2.3).

2.2 Presburger Sets

In our library, Presburger sets are unions of basic sets. For example, the set $\{x : (\exists\, q, x = 2q)\} \cup \{x : \exists\, q, x = 3q)\}$ is the union of two basic sets and contains numbers that are either multiples of 2 or multiples of 3. This can also be viewed as allowing ORs of constraints; for example, the above set can be written as $\{x : (\exists\, q, x = 2q) \vee (\exists\, q, x = 3q)\}$. FPL stores Presburger sets as a vector of basic sets of which the Presburger set is a union.

2.3 Set Operations

Our library supports a core set of operations on Presburger sets: union, intersect, subtract, complement, coalesce, equality checks, and emptiness checks. The implementations of the union and intersect operations consist entirely of copying around basic sets and constraints and are therefore

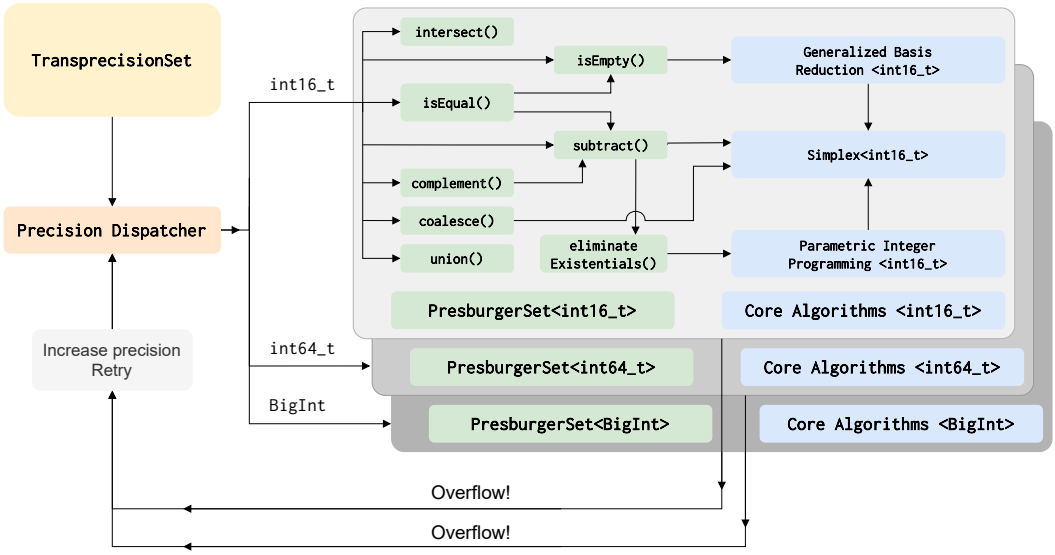


Fig. 1. The architecture of FPL. The implementations of the core operations and algorithms are templated on the integer type to be used. `TransprecisionSet` is a wrapper that dispatches to the version of our library with the correct precision. When an overflow occurs at the specified precision, an exception is thrown. This is caught by `TransprecisionSet`, which then retries the operation at a higher precision level. Our actual implementation also has a level where `__int128_t` is used.

quite amenable to vectorization. The complement operation is implemented as a subtraction, and the equality check is implemented using subtractions and emptiness checks (Figure 1).

The subtract operation makes use of the functionality for eliminating existential variables, which is based on our implementation of the Parametric Integer Programming algorithm of Feautrier [1988]. The emptiness check depends on our implementation of the Generalized Basis Reduction algorithm of Lovász and Scarf [1992]. We also implement the integer set coalescing algorithm described by Verdoolaege [2015]. All these algorithms crucially depend on support for linear programming with the ability to iteratively add constraints and roll back to an earlier state, which is provided by our implementation of the Simplex algorithm based on Simplify [Detlefs et al. 2005].

Optimizing the Simplex implementation is crucial since most of the operations depend on it. In fact, 51% of the runtime of our library is spent in Simplex. The main performance bottleneck of the Simplex algorithm in turn lies in the *pivot* operation. In our implementation, a pivot essentially consists of a sequence of row operations on a matrix of integers and can therefore be vectorized.

3 THE ARCHITECTURE OF FPL

FPL uses library-level transprecision computing to run on vectorized 16-bit arithmetic in the common case and to safely fall back to wider integer types when required, all while minimizing the dispatch overhead of switching between precision levels.

3.1 Library-Level Transprecision Computing

In FPL, the implementations of the core algorithms and set operations are oblivious to transprecision (Figure 1). Instead, they abstract over the integer type to be used, which is taken as a C++ template

parameter. All computations are performed using overflow-checked arithmetic with the specified integer type.[2] Whenever an overflow occurs, an exception is thrown.

Transprecision computing is supported via a wrapper class `TransprecisionSet` which manages the integer type to be used and dispatches to the appropriate instantiation of FPL. This class contains an `std::variant` holding a `PresburgerSet<int16_t>`, a `PresburgerSet<int64_t>`, a `PresburgerSet<__int128_t>`, or a `PresburgerSet<BigInt>`. It dispatches calls to the appropriate `PresburgerSet<Int>` instantiation of the templated library, guarded by a `try/catch` block. If an overflow exception is thrown, `TransprecisionSet` copies the currently held sets in the operands to a `PresburgerSet<WiderInt>` and retries the operation. This process can potentially continue until it switches to `BigInt`, after which no more overflows are possible.

We find that overflows are quite rare – 90% of test cases work with 16-bit integers and never overflow. Moreover, an overflow can only occur a few times in the course of a `TransprecisionSet`'s lifetime before it switches into arbitrary precision arithmetic. Therefore, the cost of throwing an exception turns out to be negligible.

By building a new Presburger library in C++, we are able to incorporate the precision to be used as a template parameter to the whole library. Such an approach would be difficult to incorporate into any existing library. In particular, this would be difficult to do manually in a C library like isl. One interesting direction of future research is to investigate whether transprecision computing could be applied automatically by the compiler, perhaps by annotating classes where this technique should be used. We hope to motivate further investigations by showcasing our results from using library-level transprecision computing for Presburger arithmetic in FPL.

FPL also works well with the realities of modern systems. We use memory-efficient data structures such as LLVM's `SmallVector`, which holds some inline memory and uses this to store its elements. It allocates dynamic memory only when the size of the vector overflows this inline memory. FPL also avoids pointer chasing, which is common in isl. We find that FPL makes 40% fewer calls to memory allocation primitives than isl.

3.2 Dispatch Overhead Is Minimized

The advantage of our design is that we execute a completely different code path for each integer type, and we are therefore able to support transprecision computing with minimal overhead. In a matrix-level or element-level implementation of transprecision computing, we have to continually dispatch into a precision-specific code path at a low-level, i.e., whenever we interact with a matrix or an integer, respectively. Moreover, whenever two transprecision objects interact with each other, such as in a binary operation involving two integers in element-level transprecision, it is necessary to bring both the objects to the same precision level before dispatch. In our library-level transprecision implementation, the objects we have to bring to the same precision are `TransprecisionSets`, and this only occurs once per high-level set operation, rather than being done for each low-level integer operation. Moreover, we only have to dispatch once per high-level operation, after which each integer type follows a completely separate code path and doesn't incur any dispatch overhead to support transprecision.

Our transprecision design is nearly optimal for the common case where the test case can be run entirely using 16-bit integers. A roofline for the common case would be a library that only supported cases that can be run with 16-bit integers and returns an error if an overflow occurs. The only overhead of FPL over such a library is a single branch per high-level set operation in the wrapper class `TransprecisionSet` to switch into the correct held alternative in the `std::variant`.

[2]Except in the case of `BigInt`, where no overflow checks are needed.

Table 1. The AVX-512 instruction set is not complete, but the available instructions depend on the bitwidth of the elements they process. Only for 16-bit elements does AVX-512 have native instructions for computing the high-bits of a multiplication as well as element-wise saturation – both important components for overflow checks.

Instruction	8-bit	16-bit	32-bit	64-bit
SIMD multiplication	✗	✓	✓	✓
SIMD multiplication - high-bits	✗	✓	✓	✗
SIMD addition	✓	✓	✓	✓
SIMD addition - with saturation	✓	✓	✗	✗

3.3 Vectorization

We use `int16_t` as our starting integer type as this enables efficient vectorized operations with overflow checks. Our library targets modern hardware with support for AVX-512 instructions. However, even in AVX-512, only 16-bit integers have native instructions for both computing the high bits of vector multiplication as well as element-wise saturated addition (Table 1, [Grosser et al. 2020]), which are important components of overflow-checked vectorized operations. In particular, computing the high bits of multiplications when each element is an 8-bit integer is not supported, which makes it difficult to use vectorized overflow-checked arithmetic with 8-bit integers.

We use LLVM's support for OpenCL vector extensions to implement overflow-checked vector arithmetic. LLVM IR supports intrinsics for vectorized operations with overflow checks. However, these are not exposed by the clang frontend. In an initial phase of development, we used a patched version of clang to expose these. We found that when targeting AVX-512, these intrinsics get lowered using instructions for saturated additions and getting the high bits of multiplications. We currently implement vectorized overflow checks directly in our C++ code by leveraging the C API to the Intel AVX-512 intrinsics. We use this API to perform saturated addition and get high bits of multiplication on our OpenCL vectors, since these operations are not directly supported. We expect that this could be implemented portably without performance loss by using LLVM IR's intrinsics and letting the compiler lower it appropriately. For example, the same approach could be supported in ARM SVE, which supports saturated addition and computing high bits of multiplication for all bitwidths we consider.

We target our vectorization efforts on computations using matrices of 16-bit integers having at most 32 columns, as a single AVX-512 vector register can hold an entire row from such a matrix. This is justified by our evaluation, which shows that in our benchmark suite, over 74% of isl's runtime is spent on test cases that we can compute using 16-bit integers and matrices with at most 32 columns.

Our library-level transprecision approach makes it possible to vectorize the copying of constraints, since a constraint is just a vector of integers. As we will see, most of the algorithms used in FPL involve quite a bit of constraint copying (Section 4). Note that in an element-level transprecision approach, each element of the vector would be a transprecision integer, probably implemented as a union or variant whose possible types must include an arbitrary-precision integer type. Therefore, vectorization would not be possible in the element-wise transprecision approach.

4 ALGORITHMIC FOUNDATIONS

We describe FPL's algorithmic design, detailing the algorithms used to implement each of the core Presburger set operations. Moreover, we provide a performance analysis of each algorithm, describing their worst-case asymptotic time complexity, the expected behavior in practice, and optimization opportunities such as vectorization.

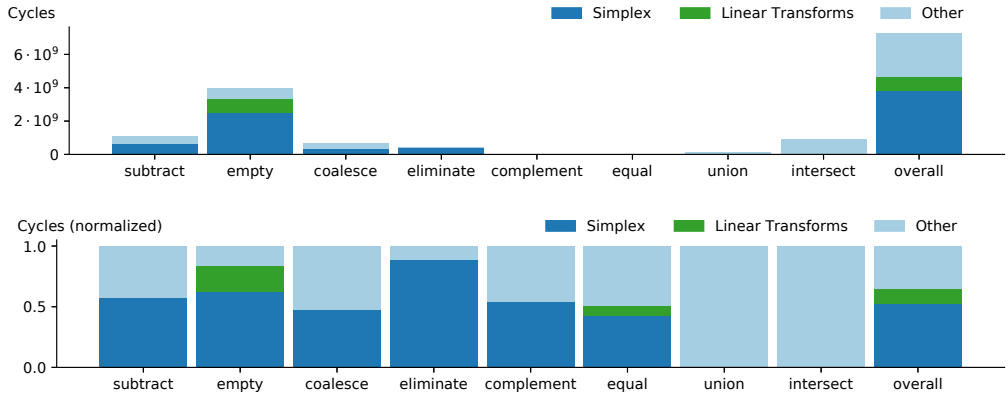


Fig. 2. Fraction of runtime spent in Simplex and in computing and applying linear transforms. These are the main two modules used in our algorithms (Section 4), and they account for less than 63% of the total runtime, indicating that a lower-level approach that applies transprecision computing to such specific submodules may miss out on accelerating a large fraction of the runtime.

To aid our performance analysis of the different set operations, we run FPL on our benchmark suite (Section 6.1) and measure how much time is spent in the key modules (Figure 2) like the Simplex implementation (Section 4.1.2) and linear transforms (Section 4.2.4).

FPL implements key algorithms from operational research and polyhedral compilation. Our algorithmic choices are inspired by work on parametric integer programming by Feautrier [1988], the use of Simplex for program analysis in Simplify [Detlefs et al. 2005], as well as various polyhedral libraries [Kelly et al. 1996; Loechner 1999], in particular, the integer set library by Verdoolaege [2010]. While Section 7 details this relationship, we focus now on the algorithmic design and concrete implementation of the library.

4.1 Auxiliary Operations

The library provides a number of auxiliary operations that are important building blocks for the implementation of higher-level operations on Presburger sets.

4.1.1 Bringing Basic Sets to a Common Space. Before computing binary operations on two basic sets, we must bring the basic sets into a common space of variables. A binary operation on two sets is only well-defined if both the sets have the same symbolic and ordinary variables. However, as mentioned before, the existential and division variables of the two sets are considered to be different. For example, the intersection of $\{x : \exists p, x = 2p\}$ and $\{x : \exists p, x = 3p\}$ is represented as $\{x : \exists p_1, p_2, x = 2p_1 + 0p_2 \wedge x = 0p_1 + 3p_2\}$, where we show the zero coefficients for clarity. We implement this by lifting the two sets to a space that has the existential and division variables of both the sets. In this case, $\{x : \exists p, x = 2p\}$ becomes $\{x : \exists p_1, p_2, x = 2p_1 + 0p_2\}$ and $\{x : \exists p, x = 3p\}$ becomes $\{x : \exists p_1, p_2, x = 0p_1 + 3p_2\}$. We can then run our intersection algorithm assuming the variables in both the sets to be the same.

In general, the common space of two basic sets is the one created by taking the common ordinary and symbolic variables, and including separately the existential and division variables from both the sets. The constraints in the first set use only variables that were originally in the first set – as in the example, their coefficient lists will be padded to have zero coefficients for all the existential and division variables that were added from the second set. Similarly, the constraints of the second set have zero coefficients inserted for all the existential and division variables from the first set.

Performance Analysis. This operation essentially inserts some zero-initialized columns into a matrix, where the rows of the matrix are constraints and the columns are dimensions. This takes linear time. As we will see, for most operations it is preferable to store the constraint matrix in row-major form, so there is little opportunity for vectorization here.

4.1.2 Simplex. Our library uses an implementation of the Simplex algorithm based on ideas from Simplify [Detlefs et al. 2005]. This algorithm can be used to query properties of basic sets using linear programming, such as computing the maximum and minimum values of an affine expression subject to the constraints of a basic set, detecting rational redundancies in constraints, finding a rational sample point, and checking for rational emptiness. A subset of constraints is rationally redundant if removing these constraints does not introduce any new rational solutions. A rational sample point is a rational solution to the constraints in the set, and a basic set is rationally empty if it contains no rational sample point. Computing such rational properties is done using the Simplex algorithm, which is efficient in practice [Schrijver 1986; Shamir 1987]. We found that we never required more than 115 pivot steps to solve any linear programming problem that arises in our entire benchmark suite (Figure 15).

The Simplify-based implementation also supports incrementally adding constraints, taking snapshots, and rolling back changes to a prior snapshot. This capability is used in the implementations of the Parametric Integer Programming [Feautrier 1988] and Generalized Basis Reduction [Lovász and Scarf 1992] algorithms, as well as the subtraction algorithm.

Performance Analysis. The Simplex algorithm can take exponential time in the worst case. However, it is known to be quite efficient in practice. The main bottleneck of the Simplex algorithm lies in the *pivot* operation. With an appropriate choice of internal representation, the main bottleneck of the pivot function can be written as a series of row operations on a matrix. The Simplex algorithm therefore benefits greatly from vectorization [Grosser et al. 2020].

4.1.3 Eliminating Existential Variables. The algorithm for subtraction does not directly support sets containing existential variables, but does support sets containing division variables. In this section, we describe an algorithm that, given a basic set containing existential variables, finds a representation of the same set that does not involve any existential variables, though it may involve division variables. For example, $\{x : \exists q, x = 6q\}$ is actually the set of multiples of 6. We can represent this set as $\{x : x = 6\lfloor x/6 \rfloor\}$, which does not involve any existential variables though it does involve division variables.

We eliminate existential variables using the Parametric Integer Programming (PIP) algorithm of Feautrier [1988]. Given a basic set where some variables are tagged as parameters and the rest are non-parameters, such that the non-parameters are all constrained to be non-negative, PIP can compute the set of values of the parameters such that there exists an assignment to the non-parameters satisfying the constraints. This set of values is represented as a basic set in the space of parameters. The representation of this basic set may involve division variables, but will not involve any existential variables. The implementation of PIP internally uses the Simplex implementation described in Section 4.1.2.

We eliminate existential variables from a basic set by running PIP on it, setting the existential variables as non-parameters and setting the other variables as parameters. PIP returns a Presburger set corresponding to the set of values of the variables other than the existential variables such that there exists some satisfying assignment to the existential variables, which is therefore another representation of the same basic set, and one that does not contain any existential variables. However, this may not work if some existential variables can attain negative values since the PIP algorithm as described by Feautrier [Feautrier 1988] does not support this case.

We support negative-valued existential variables by transforming each existential variable e_i to $e_i + M$, where M is a newly added variable. We consider M to have a value of infinity in our implementation of the PIP algorithm, so that our new existential variables $e_i + M$ are guaranteed to be non-negative. If the original variables were $x_1, \ldots x_n, e_1, \ldots e_m$, then after performing the transform the variables become $x_1, \ldots x_n, e_1 + M, \ldots e_m + M, M$.

Performance Analysis. The PIP algorithm involves running two instances of Simplex in parallel and spends most of its time in these (Figure 2), so it benefits greatly from optimizations to Simplex. Its runtime is exponential in the worst case since it uses Simplex.

4.2 High-Level Operations

Our library provides a set of high-level operations on Presburger sets: union, intersect, subtract, complement, sample, equality, and coalesce. In this section, we describe the algorithms used to implement these operations. Throughout this discussion, we will use d to refer to the total number of variables in the common space of all the basic sets involved.

4.2.1 Union. The union of two Presburger sets S and T is a set whose list of basic sets is the concatenation of the lists of S and T.

Performance Analysis. If the list of basic sets is represented as a list of pointers to basic sets, one could use a copy-on-write optimization and refrain from copying the basics sets themselves until necessary. This approach is used by isl. However, this induces an overhead in terms of pointer chasing and reference counting, which FPL avoids. Moreover, we find that the fraction of time spent in the union operation is negligible (Figure 9).

FPL stores the basic sets as a vector of objects. If the operands are temporary values that will not be reused, we can move the basic sets instead of copying them. When data structures with inline memory are used, moving has a relatively lower performance benefit. Since union forms a very small fraction of the total runtime, it is still worthwhile to use such data structures.

4.2.2 Intersect. We represent the intersection of two Presburger sets S and T as the union of intersections of basic sets. Let $S = \bigcup_i S_i$ and $T = \bigcup_j T_j$ be Presburger sets which are the unions of basic sets S_i and T_i respectively. Then we have $S \cap T = \left(\cup_i S_i\right) \cap \left(\cup_j T_j\right) = \cup_i \cup_j \left(S_i \cap T_j\right)$. To intersect two basic sets S_i and T_j, we first bring them to a common space (Section 4.1.1). The intersection of two basic sets that lie in the same space is simply the basic set that satisfies both lists of constraints, i.e., a basic set whose lists of inequalities and equalities are the concatenations of the respective lists.

Performance Analysis. Let $n(S), c(S)$ and $n(T), c(T)$ be the number of basic sets and the maximum number of constraints in any basic set in S and T respectively. The result has $n(S)n(T)$ basic sets and $n(S)c(S)n(T) + n(T)c(T)n(S)$ constraints, so the total number of integer coefficients in the result is $O(d \cdot [n(S)c(S)n(T) + n(T)c(T)n(S)])$. Constructing the result therefore takes quadratic time. Since intersection mostly involves copying around constraints between sets, this can easily benefit from vectorization.

4.2.3 Subtract and Complement. We introduce subtract and complement together, as their implementations depend on each other. We first discuss how to handle basic sets without existentials or divisions, then discuss how to handle existentials and divisions in basic sets, and finally discuss how to subtract and complement full Presburger sets.

Basic Sets without Existentials or Divisions. The complement of a basic set is the union of multiple basic sets, one for each constraint. The basic set for each constraint corresponds to the region where

that constraint is the first constraint violated. Consider the basic set $\{(x, y) : x \geq 0 \wedge y \geq 0\}$. The subtraction algorithm represents the complement of this set as $\{x : (x \leq -1) \vee (x \geq 0 \wedge y \leq -1)\}$. We have two basic sets here; the first is the set of points not satisfying the first constraint, and the second is the set of points satisfying the first constraint but not the second. One could also represent the result set as $\{x : (x \leq -1) \vee (y \leq -1)\}$. However, the former approach is preferred as this produces basic sets that are smaller (have fewer points), which in turn makes it more likely that the heuristics detailed below get triggered.

The complement of the set of solutions to $a_1x_1 + \ldots + a_nx_n + c \geq 0$ is just the set of solutions to $a_1x_1 + \ldots + a_nx_n + c \leq -1$, since the variables are integers. Similarly, for equalities, the complement of the set of solutions to $a_1x_1 + \ldots + a_nx_n + c = 0$ is the set of solutions to $(a_1x_1 + \ldots + a_nx_n + c \leq -1) \vee (a_1x_1 + \ldots + a_nx_n + c \geq 1)$.

In general, let $B = \bigcap_i B_i$ be a basic set without existential or division variables, where each B_i is the set of solutions to a single inequality. We write the complement B^c as the union of the parts B_1^c, $B_1 \cap B_2^c$, $B_1 \cap B_2 \cap B_3^c$ and so on. Here the first disjunct, B_1^c is the set of points that do not satisfy the first constraint. The second disjunct, $B_1 \cap B_2^c$ is the set of points that satisfy the first constraint but not the second. Similarly, the ith disjunct is the set of points that satisfy the first $i - 1$ constraints but not the ith. Since this is a partition of the invalid points according to the first constraint they violate, this expression is equal to B^c. This also allows us to subtract basic sets, since for basic sets A and B, $A \setminus B = A \cap B^c$.

Basic Sets with Existentials or Divisions. Consider the set $\{x : x = 7\lfloor x/7 \rfloor\}$. It is clear that the complement of this set is $\{x : x \neq 7\lfloor x/7 \rfloor\}$, even though divisions are implemented in Presburger arithmetic by adding extra existentially quantified variables. Therefore, we handle divisions by treating the division variables as be ordinary (unquantified) variables and computing the complement of the set considering only the inequalities and equalities and ignoring the division constraints. This is easy since we maintain the divisions separately from the other constraints. The final internal representation of the result set will of course again use existential quantification and division variables to represent divisions, but these will be the same divisions as in the input basic sets, so we can simply copy the divisions from the input basic sets to the result basic sets.

If B involves existential variables, we eliminate them first (Section 4.1.3) to obtain a Presburger set whose basic sets do not involve any existential variables (but may involve divisions), which we handle using the algorithm for Presburger sets detailed below.

Presburger sets. We subtract a Presburger set $T = \bigcup_j T_j$ from a Presburger set $S = \bigcup_i S_i$ by subtracting the individual basic sets of T one after another. We have $S \setminus T = \bigcup_i (S_i \setminus T)$. We compute $S_i \setminus T$ by subtracting each basic set of T in succession, i.e., we first compute $S_i \setminus T_1$; this produces a union of basic sets, from which we subtract T_2. We keep doing this recursively until there are no parts of T left to subtract, at which point we obtain $S_i \setminus T$. We then compute $S \setminus T$ as the union of all the $S_i \setminus T$. The complement of a Presburger set A is computed as $U \setminus A$, where U is the set of all points in the space of A.

If T is made of $n(T)$ basic sets and each set has c constraints, then the result could have up to $c^{n(T)}$ basic sets in the worst case, an exponential blowup. Since the subtraction $S_i \setminus T$ is performed recursively, pruning some branches of computation early can make a significant difference. Let subtract(B, j) be a recursive function that computes $(B \setminus T_j) \setminus T_{j+1} \ldots$, with the initial call being to subtract$(S_i, 1)$. At every recursive call, we apply the following two heuristics:

Heuristic 1: If $B \cap T_j$ contains no rational points, then $B \setminus T_j = B$, so we immediately return subtract$(B, j + 1)$.

Heuristic 2: Check which constraints of T_j are rationally redundant given the constraints of B. No point in B can violate such a constraint, so the basic set corresponding to the region where this constraint is violated is empty and can be skipped. This can significantly cut down the number of basic sets produced.

The Simplex implementation can check these properties quite efficiently in practice, as mentioned in Section 4.1.2. To further minimize the cost of these optimizations, we maintain a Simplex object corresponding to the set B. Whenever B is modified, we incrementally track the changes in the Simplex object. When we return from a recursive call, we roll back the Simplex to a snapshot taken at the start of the call so that the caller can continue using it. This saves the cost of constructing a Simplex object from scratch for each recursive call.

Performance Analysis. Let $n(S), c(S)$ and $n(T), c(T)$ be the number of basic sets and the maximum number of constraints in any basic set in S and T respectively. The computed representation of $S_i \setminus T$ could have up to $n(S)c(T)^{n(T)}$ basic sets in the result. Each result basic set could have up to $c(S) + c(T)n(T)$ constraints, so the result can have $O(d \cdot n(S) \cdot c(T)^{n(T)}(c(S) + c(T)n(T)))$ integer coefficients, making the worst-case runtime exponential in the input size. The heuristics may improve the worst-case number of basic sets and constraints. However, our implementation uses the Simplex algorithm which can be exponential in the input size in the worst case, so we do not gain in terms of asymptotic complexity.

FPL's implementation of the subtraction algorithm spends around half of its time in Simplex (Figure 2). The other operations it performs are adding constraints to basic sets, intersections, and unions. Except for the union, all of these can be accelerated using vectorization in our library-level transprecision approach.

4.2.4 Sample. In this section, we describe an algorithm to check if Presburger sets are empty and find a sample point in the set if one exists. To find a sample in a Presburger set, we search each basic set until we either find a sample or have searched all the basic sets. We look for a sample in a basic set using a branch and bound algorithm, which computes each variable the range of rational values it can take. It then recursively tries all integer values within this range. Since this can take exponential time in the worst case, we use the Generalized Basis Reduction (GBR) algorithm to transform the variables such that the range of values of the variables in the transformed set is reduced. It is also possible to implement the sampling operation using the method of cutting planes [Schrijver 1986]. However, based on our experience with isl, we believe that the GBR-based approach is likely to have better performance.

The GBR algorithm is only applicable to bounded sets, so for unbounded sets we require a different algorithm. We check if the set is bounded or unbounded using our Simplex implementation (Section 4.1.2) and dispatch to the appropriate algorithm.

Bounded Sets. For bounded sets, we use the branch and bound algorithm along with the GBR algorithm. The branch and bound algorithm checks, for each variable, the maximal and minimal rational values that this variable can attain. These values can be computed by linear programming, as discussed in Section 4.1.2. For each possible integer value in this interval, we fix the value of this variable and recursively continue to the next variable. If at any point, a variable doesn't have any possible integer values, we backtrack. If we find an assignment to all the variables, then this is the required sample point. Otherwise, the set is empty.

The GBR algorithm transforms the variables such that they have a smaller range of possible values, as mentioned before. The implementation of this algorithm requires linear programming as well as the ability to incrementally add constraints and roll back changes. This functionality

is provided by the Simplex implementation. Since GBR doesn't support unbounded sets, some additional processing is required in that case.

Unbounded sets. For unbounded sets, we first perform a variable transformation such that a prefix of the variables – say, the first k variables – are bounded, and the remaining variables are unbounded. We then create a bounded set by projecting out the unbounded variables. It can be shown that no combination of constraints involving unbounded variables can produce a constraint on the bounded variables, so this can be accomplished by simply discarding all constraints that have non-zero coefficients for unbounded variables. Finally, we substitute the values found for the bounded variables back into the original set. This satisfies all the constraints except those that involve unbounded directions. Since all the bounded variables have been substituted away, this set is a full-dimensional cone, a cone that is unbounded along every axis. Such a cone is certainly non-empty; it is only left to find a sample point in the cone.

Full-dimensional cones. We find a sample in the full-dimensional cone by shrinking the cone, finding a rational point in the cone using Simplex, and rounding this point up to the nearest integer point. We shrink the cone enough such that a rational point inside the shrunken cone, rounded up to an integer point, will always lie within the original cone. Since the cone is unbounded along every direction, this shrinking cannot make it empty. Hence, we will always find a point in the shrunken cone.

To shrink the cone, we tighten each constraint. Decreasing the constant term of an inequality tightens it, since for an expression constrained to be greater than or equal to zero, decreasing the constant term causes previously borderline but valid points to become invalid. We want to ensure that if $(x_1, \ldots x_n)$ satisfies the tightened constraint, then this point rounded up satisfies the original constraint $a_1x_1 + \ldots a_nx_n + c \geq 0$. After rounding up, the new value of x_i will be in the range $[x_i, x_i + 1]$. (In fact, it can never be $x_i + 1$, but it is more convenient to work with this closed interval.) In terms of satisfying the constraint, the worst case occurs when x_i increases by one for all i such that a_i is negative and all the other x_i stay unchanged. Let d be the sum of the magnitudes of the negative a_i. The rounded point surely satisfies the original constraint if the original rational point satisfied $a_1x_1 + \ldots + a_nx_n + c - d \geq 0$, so we use this as the tightened constraint.

Performance Analysis. The algorithm for integer linear programming based on the Generalized Basis Reduction algorithm is known to run in polynomial time when the number of dimensions is fixed, if linear programming is implemented using a polynomial-time algorithm. Our implementation uses the Simplex algorithm, which is efficient in practice even though it is known to take up to exponential time in the worst case.

FPL's implementation of the emptiness check algorithm spends 62% of its runtime in Simplex and Generalized Basis Reduction, and 22% of its time in computing and applying the transform used in the case of unbounded sets to make a prefix of the variables unbounded and the rest bounded. Applying the transform amounts to a matrix multiplication, which is non-trivial to vectorize. *Computing* the transform is done essentially using a series of column operations, similar to bringing a matrix to column echelon form, which could be vectorized if the matrix is stored in column-major form. The Simplex algorithm also benefits greatly from vectorization, as mentioned earlier.

4.2.5 Equality. Let S and T be two Presburger sets. $S = T$ iff both $S \subseteq T$ and $T \subseteq S$, and $S \subseteq T$ iff $S \setminus T$ is empty. Therefore, S is equal to T iff both $S \setminus T$ and $T \setminus S$ are empty.

4.2.6 Coalesce. The coalescing algorithm [Verdoolaege 2015] is a heuristic that, given a Presburger set made up of many basic sets, tries to find a simplified representation using as few basic sets as

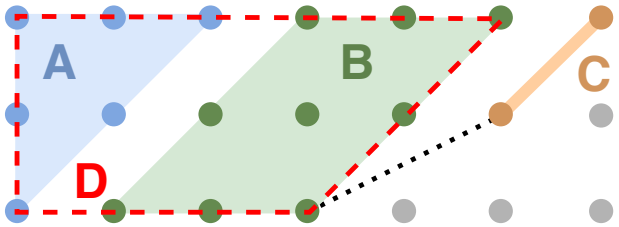


Fig. 3. The union of the basic sets A, B, and C is in fact itself a convex object and can be represented as a single basic set. Computing a representation of $D = A \cup B$ as a basic set is straightforward, but doing so for $C \cup D$ requires the more involved wrapping subroutine.

possible. Since the running time of many algorithms depends on the number of basic sets, this can be a significant optimization.

Consider the Presburger set $S = A \cup B \cup C$ as in Figure 3. The coalescing algorithm considers every pair of basic sets in the union and tries to coalesce them into a single basic set. In our example, it is able to detect that $A \cup B$ is convex. In this case, we obtain a representation of this union as a basic set by simply making a basic set D with all the constraints in A and B except those corresponding to the sides that are adjacent to the other set. This gives us the simplified representation $S = C \cup D$.

In fact, $C \cup D$ is also convex, but it is not possible to represent this union as a basic set by simply selecting a subset of the constraints of C and D.

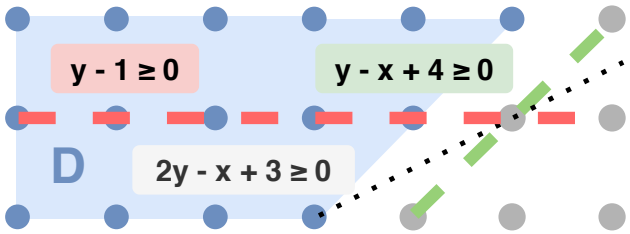


Fig. 4. The wrapping algorithm computes the required constraint by interpolating between a constraint that holds for all points in D, $y - x + 4 \geq 0$, and one that does not, $y - 1 \geq 0$. In this case, an equal mix of both the constraints, $(y - x + 4) + (y - 1) \geq 0$, i.e., $2y + 3 \geq x$ is the required constraint.

Wrapping. To compute the constraint wrapping C and D together, we interpolate between a constraint that holds for all points in D and one that does not hold for all points in D, mixing the two constraints in the right proportion to produce the sharpest constraint that holds for D. In the example (Figure 4), the constraint $y - x + 4 \geq 0$ holds for D and $y - 1 \geq 0$ does not hold for all points in D. The details of how we choose the constraints to interpolate between are out of the scope of this section and can be found in the original paper. The required constraint, shown by the black dashed line, is given by interpolating between the two constraints in equal proportion to get $(y - x + 4) + (y - 1) \geq 0$, i.e., $2y - x + 3 \geq 0$.

Performance Analysis. The coalesce algorithm compares every pair of basic sets, so the number of times it uses linear programming is quadratic in the number of basic sets. Since we implement linear programming using the Simplex algorithm, its overall time complexity is exponential. FPL's implementation of the coalesce algorithm spends 47% of its runtime in Simplex (Figure 2).

5 PRESBURGER SETS IN LLVM AND MLIR

We aim to support Presburger arithmetic natively in LLVM/MLIR by contributing our work to the LLVM repositories. Large parts of FPL have already been upstreamed to LLVM's MLIR project, where the library is deeply integrated into the compiler. For example, MLIR uses a class `IntegerSet`, to represent the basic sets that correspond to the IR that the compiler is operating on. These `IntegerSet` objects are quite close to the IR and are represented as a kind of expression tree. To perform analyses on these sets, we require the set to be represented as a list of constraints (Section 2). MLIR provides support for "flattening" these `IntegerSet` expression trees into what are called `FlatAffineConstraints`, which represent basic sets in the form we require.

We have upstreamed support for Presburger sets, which are unions of basic sets. We provide support for a number of standard operations on integer sets in MLIR, including union, intersect, subtract, complement, equality checks, and integer emptiness checks. This support is provided directly within MLIR, and operates natively on the objects that MLIR uses to perform analysis. These objects can readily be generated from the IR, providing a smooth flow from the compiler to the analysis operations provided by FPL.

We fully support divisions in all these operations; our next steps will be to provide support for coalescing and arbitrary existentially quantified variables. Finally, we will explore with the LLVM community how to upstream performance optimizations such as SIMDization as well as transprecision computing. In particular, we will need to replace our use of SIMD intrinsics with a more production-quality and portable SIMD implementation. Also, we need to find an alternative way to implement transprecision computing that works in LLVM's default configuration, where exceptions are disabled. While optimizing our implementation will require further thought, we expect that the upcoming availability of a Presburger library within MLIR will make it significantly easier to develop new Presburger-based tools in the MLIR ecosystem.

6 EVALUATION

We evaluate FPL on a new benchmark suite that we derive from real-world compilers. We present the first comprehensive suite of Presburger arithmetic test cases representative of the computations in polyhedral compilers. We then evaluate both our correctness and performance against the state-of-the-art and perform a detailed analysis of the performance properties of FPL.

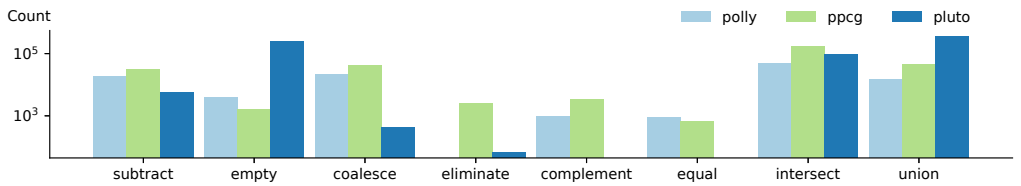


Fig. 5. Distribution of test cases for each operation extracted from each compiler.

6.1 Presburger Set Benchmarks

Our benchmark suite characterizes the usage of Presburger arithmetic in polyhedral compilers and provides a comprehensive set of runtime benchmarks suitable for performance evaluating a library for Presburger set arithmetic. To the best of our knowledge, there is not yet any benchmark suite that comprehensively characterizes the problem instances that are typical in polyhedral compilation. We create the benchmark suite by running Polly[3] [Grosser et al. 2012], the PPCG GPU compiler

[3]We use commit `bfdafa32` from https://github.com/llvm/llvm-project.

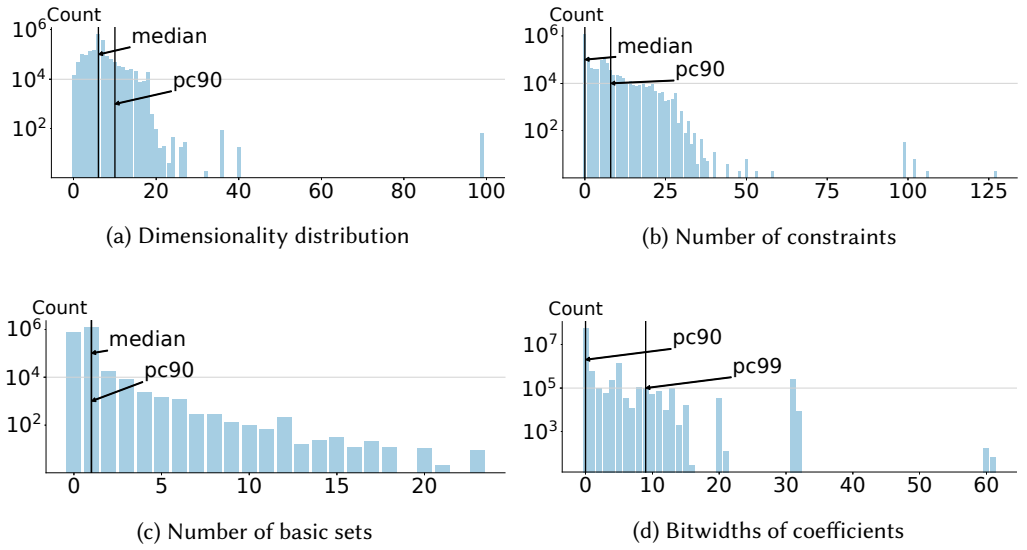


Fig. 6. 90% of sets occurring in our benchmark suite have at most eight dimensions, at most eight constraints, and at most one basic set. 90% of constraint coefficients are zero, and 99% fit in 9 bits.

0.08.2 [Verdoolaege et al. 2013], and Pluto[4] [Bondhugula and Ramanujam 2007] on Polybench 4.1 [Pouchet 2012] and extracting data on the Presburger set operations they perform. In particular, we extract the inputs and outputs for core operations on Presburger sets: union, intersect, subtract, complement, coalesce, equality checks, emptiness checks, and eliminating existentials (Figure 5). These operations cover the various tasks that are typical in polyhedral compilation, such as SCoP construction, dependence analysis, scheduling, and AST construction. Our benchmark suite hence provides a faithful representation of Presburger arithmetic as used in polyhedral compilation.

In total, we extract 1,140,039 test cases. Out of these, there are 10,800 coalesce test cases involving sets of rational points. While these could be supported without major changes to the architecture or data structures, we currently only support sets of integer points. As such, we do not consider these cases. This leaves us with 1,129,239 test cases: 111,368 from Polly, 305,041 from PPCG, and 712,830 from Pluto (Figure 5).

We characterize the properties of typical Presburger sets occurring in polyhedral compilers. The sets are typically small in terms of dimensionality, number of constraints, number of basic sets, and size of coefficients. The median dimensionality (Figure 6a) of the sets we consider is only 8 dimensions, with 99.9% of sets having less than 20 dimensions. The number of constraints (Figure 6b) is, excluding a large number of empty and universe zero-constraint sets, typically in the low tens and always below 200. 90% of Presburger sets (Figure 6c) have only one basic set, and 99% of constraint coefficients fit within 9 bits. In fact, 90% of the coefficients are zero. These characteristics indicate that optimizing for low dimensionality and small integer coefficients is likely to prove fruitful. The low dimensionality indicates that we can exploit SIMD parallelism with up to 32 lanes. Most of the operations are embarrassingly parallel across basic sets. However, since most tests have zero or one basic sets, we cannot exploit this coarse-grained parallelism very well. As a result, GPUs are likely not a good target for FPL, since we cannot utilize all the multiprocessors. Rather, the local parallelism offered by CPU SIMD instructions is well-suited to our application.

[4]We use commit 5b13ddcc from https://github.com/bondhugula/pluto.

Table 2. Test distribution across operations and kinds of sets. Subtract, complement and equal never receive inputs with existential variables, possibly because these are always eliminated before being passed to these operations.

Ordinary	Division	Existential
420,995	1,252	24
290,109	30,236	56
55,751	343	0
4,305	0	0
46,537	16,925	9
1,562	1	0
258,006	520	0
1596	946	66

Table 3. FPL computes correct results on all 1,129,239 tests (green). Some set types never occur for certain operations (grey).

	Ordinary	Divisions	Existentials
Union	100%	100%	100%
Intersect	100%	100%	100%
Subtract	100%	100%	n/a
Complement	100%	n/a	n/a
Coalesce	100%	100%	100%
Equal	100%	100%	n/a
Empty	100%	100%	n/a
Eliminate	100%	100%	100%

We find that the test set contains 50,223 test cases involving division variables, indicating that this aspect of Presburger arithmetic sees significant use in polyhedral compilation and that the test set has good coverage for it. On the other hand, no sets with existential variables occur for the subtract, complement, and equal operations (Table 2), possibly because these compilers eliminate them before passing them to these operations. In order to catch any sets where this may have occurred, we explicitly extract all the sets from which Pluto and PPCG eliminate existentials.[5]

Overall, the benchmarks exercise every aspect of the algorithmic foundations. The basic algorithms for union, intersect, subtract, empty and coalesce are tested by tens or hundreds of thousands of tests. There is comprehensive coverage of the support for division variables, as mentioned before. In particular, the 30,236 intersect test cases containing divisions provide good coverage for the functionality for bringing sets to a common space (Section 4.1.1). By also extracting test cases for eliminating existentials, we create a comprehensive benchmark and test suite for Presburger arithmetic as used in polyhedral compilation.

6.2 Correctness

We evaluate the correctness of our library on the benchmark suite across different operations and Presburger set types. For our experiments (Table 3) we use isl, a state-of-the-art library for Presburger arithmetic, as an oracle.

We find that our algorithm produces correct results for all 1,129,239 test cases for all operations demonstrating the robustness of FPL's implementations of these operations on a large real-world test set.

6.3 Performance

We analyze the performance of the individual operations extracted from different compilers, comparing our overall performance against isl. We then investigate the suitability of transprecision computing to workloads in polyhedral compilers, as well as the overhead incurred by our approach

[5]For technical reasons, we did not extract test cases for eliminating existentials from Polly.

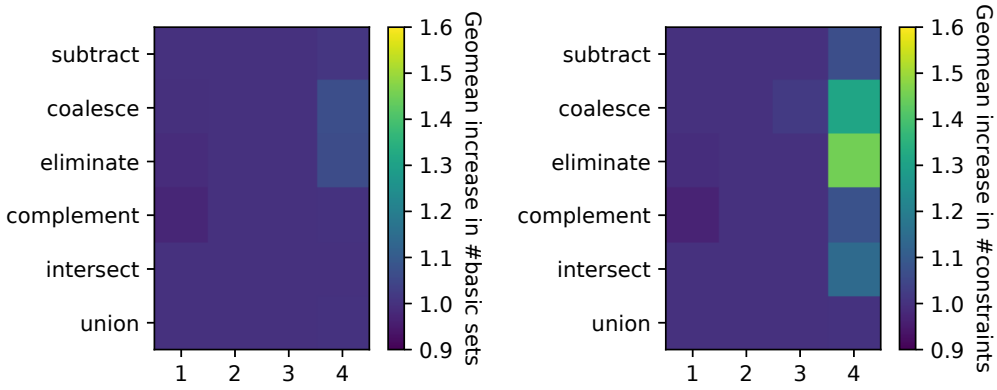


Fig. 7. Comparison of the representation sizes of output sets in FPL versus that in isl. We visualize the geometric mean of the ratio of the number of basic sets and constraints in FPL's output to that in isl's output. We find that in the fourth (worst-performing) quartile of test cases, the eliminate and coalesce operations have an output representation with a geomean increase of 1.45x and 1.3x respectively in the number of constraints. The remaining operations have output sizes comparable to isl.

to transprecision computing. Finally, we evaluate the design of our library by investigating memory allocations as well as cache behavior.

Representation size of output sets. We evaluate the performance of our library in comparison with isl, a state-of-the-art library for Presburger arithmetic. FPL has an order of magnitude speedup over isl in its default configuration which uses GMP for integer arithmetic. However, this is in part because isl has some expensive heuristics that exist solely to simplify the representations of the output sets. For example, this results in a reduction in the number of constraints in isl's output by 1.57x on average (geomean) relative to FPL. Since we have not yet performed a cost-benefit analysis of such expensive heuristics, we do not currently implement them. In order to make an apples-to-apples comparison and prevent ourselves from gaining an unfair advantage on our single-operation benchmark, we disable such output simplifications in isl's implementations of union, subtract, intersect, and complement when we measure runtimes.

Figure 7 visualizes the average (geometric mean) growth by quartile in the number of basic sets and constraints in our output as compared to isl with its heuristics removed. The emptiness checks and equality checks are not shown since they produce boolean output. When both isl and FPL have a trivial output having zero basic sets or constraints, the ratio is considered to be 1. The figure does not include test cases where isl produces a trivial output but FPL does not, since the ratio is undefined in that case (it involves division by zero). Such cases form around 18% of the subtraction outputs and less than 2% of each of the remaining operations. We believe that cases in subtraction are a result of heuristics that we did not remove because doing so would make isl slower, resulting in an unfair comparison; these heuristics both improve efficiency *and* improve the size of the representation.

We see that our output is of comparable size to isl in all other operations except eliminate and coalesce. In the worst-performing quartile, the eliminate and coalesce operations have a geomean increase of 1.45x and 1.3x respectively in the number of constraints. This is because we do not remove the additional heuristics from these operations. We do not remove heuristics in isl's functionality for eliminating existentials since it appears to be non-trivial to do so in isl's design. Moreover, we do not remove any heuristics from isl's implementation of coalescing since

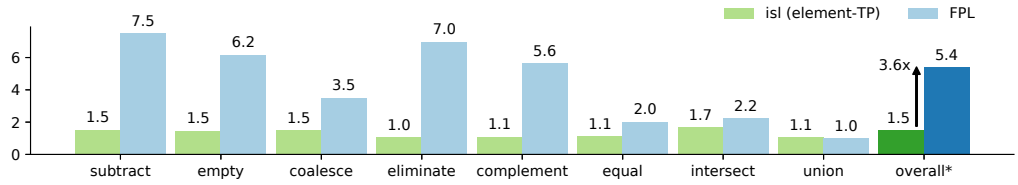
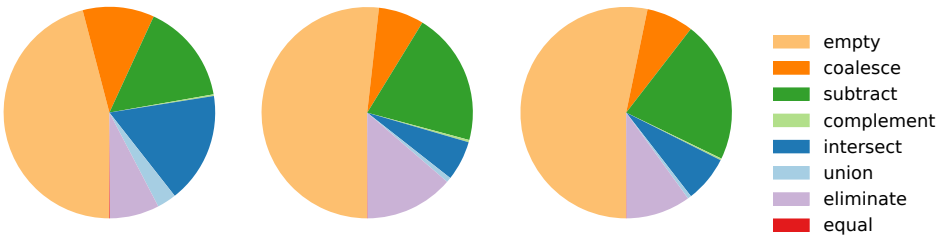


Fig. 8. Speedup of our library over isl in terms of total runtime per operation. The overall* speedup does not include eliminate and coalesce since our output representation sizes are not comparable to isl (Figure 7). We see an overall speedup of 5.4x over isl with GMP and 3.6x over isl with element-transprecision.

the purpose of this operation is to simplify the representations of sets. Since our output sizes for eliminate and coalesce are not comparable to isl, we do not include these operations when calculating our overall speedup (Figure 8).



(a) FPL (library-TP) (b) isl (element-TP) (c) isl (GMP)

Fig. 9. Fraction of total runtime spent in each operation by (a) FPL with library-level transprecision, (b) isl with element-level transprecision, and (c) isl with GMP. The time spent in complement and equality checks is negligible. Over half of isl's total runtime is in emptiness checks.

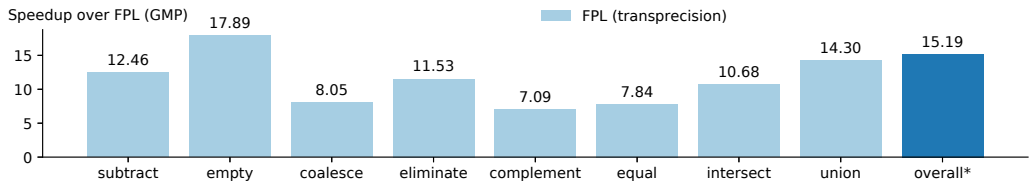


Fig. 10. Speedup of FPL with transprecision over FPL running on GMP. We find that FPL with transprecision has a 15x overall* speedup over FPL running with only GMP.

Experimental setup. We run all the test cases in succession, five times each, and consider the median execution time of each test case. Since polyhedral compilers typically run many operations in succession, this is a realistic execution environment. We execute all the benchmarks on a single thread on a test system having an Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz with an 8 MiB L3 cache and 16 GB of RAM. In particular, our system has support for AVX-512 wide vector instructions, which FPL is able to exploit. We ran all experiments with the CPU frequency locked.

Speedups. FPL shows an overall speedup of 5.4x in terms of total runtime over isl running in its default configuration with GMP-based integers and 3.6x over isl enhanced with element-level transprecision (Figure 8). This metric is of particular interest for compilers, which run many operations in sequence before returning anything to the user. In this setting, the total runtime of the operations is what is visible to the user, rather than the time taken on individual test cases. We are faster than isl with GMP on all operations except union, possibly due to isl's usage of copy on write with reference-counted sets (Section 4.2.1).

In the emptiness check operation, we show speedups of 6.2x over isl with GMP and 4.1x over isl with element-level transprecision. The emptiness check has a boolean output and a large fraction of isl's total runtime is spent in this operation (Figure 9). Thus, we achieve a speedup of 6.2x on a crucial operation where output simplifications certainly do not affect the result, and one that forms over half of isl's total runtime on the benchmark suite.

Comparing FPL running with full transprecision and FPL running on only GMP, we found that FPL with transprecision has a speedup of 15x (Figure 10) over FPL running with arbitrary precision arithmetic. Thus, transprecision computing is highly effective for our use case.

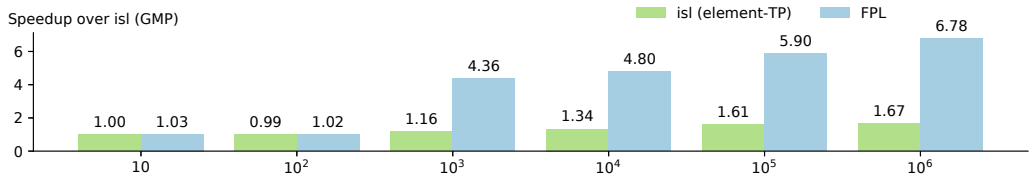


Fig. 11. We categorize the test cases into buckets according to the order of magnitude of their runtime in isl (GMP). We find that the speedup of FPL over isl (GMP) for fast-running test cases is negligible but the speedup for longer running test cases is above 4x. There are nine test cases for subtraction taking tens of millions of cycles, on which we have a speedup of 116x; this outlier bucket is not shown.

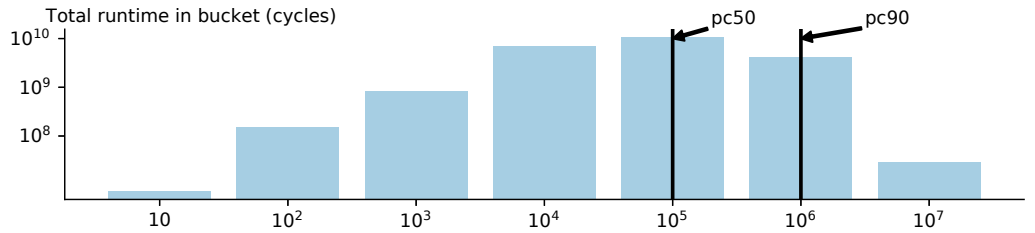
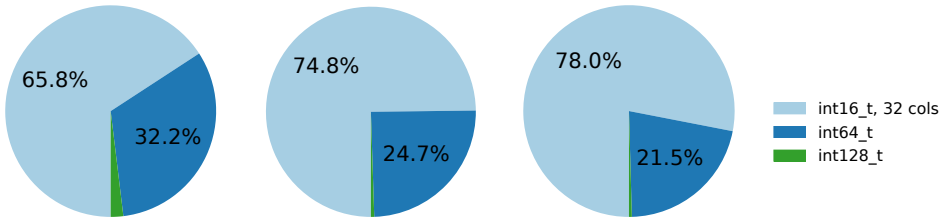


Fig. 12. So as to understand the relative importance of each bucket above, we also measure the total time spent by isl (GMP) in each bucket. We find that the 10^4, 10^5, and 10^6 buckets together account for over 90% of the total runtime.

Our speedups are derived from the longer running test cases. Categorizing the test cases by the order of magnitude of their runtime in isl (GMP), we find that FPL's speedup over isl (GMP) is negligible on the test cases that run in tens or hundreds of cycles but above 4x on the longer running test cases whose order of magnitude of runtime is in the thousands to tens of millions of cycles (Figure 11); these test cases account for over 90% of the total runtime of isl (GMP) (Figure 12).

We expect that the demonstrated performance of FPL will motivate the future implementation of higher-level polyhedral compilation algorithms using FPL. We are optimistic that such higher-level algorithms will benefit from the fast performance that FPL provides for Presburger set operations.



(a) FPL (b) isl (element-TP) (c) isl (GMP)

Fig. 13. Total runtime broken down according to the precision level FPL needs to compute the results. FPL does not require arbitrary-precision arithmetic for any of the test cases in the benchmark suite. isl spends over 74% of its time on cases that FPL can compute using just int16_t and matrices with at most 32 columns. isl spends over 99% of its time on test cases where int64_t suffices.

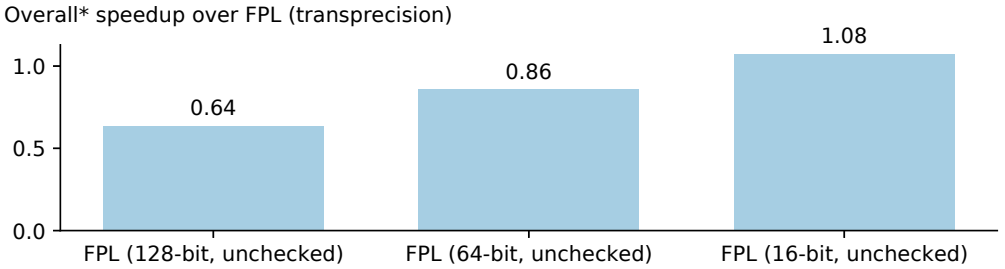


Fig. 14. Overall* speedup of FPL running at different precision levels without overflow checks, as compared to FPL with full transprecision. The runs with 16-bit and 64-bit precision were run and compared only on the subset of test cases that can be computed at these precision levels without overflow. Running with 16-bit integers and no overflow checks results in only a 1.08x speedup over FPL with transprecision, indicating that the overhead of transprecision computing is low.

Applicability of transprecision computing. We have investigated the distribution of the coefficient values in the input sets and found that these are usually small (Figure 6d). However, this is not sufficient for transprecision computing to be effective since – in theory – even small constraints could produce very large integer values during computations performed in, e.g., Simplex. To obtain a better picture of the applicability of transprecision computing to polyhedral compiler workloads, we evaluate what fraction of the total runtime is spent in test cases that can be computed using only 16-bit integer and 32-columns, which is the lowest precision mode in FPL (Section 3.3). We also check what fraction of test cases require 64-bit integers, 128-bit integers, and arbitrary-precision integers.

We find that over 99% of isl's runtime is spent in test cases that FPL computes using only 64-bit integers, and over 74% is spent in cases that FPL can run with just 16-bit integers and 32-column

matrices (Figure 13). Moreover, arbitrary-precision arithmetic is never needed. Transprecision computing is therefore highly applicable to typical workloads in polyhedral compilers.

Comparison to fixed precision without overflow checks. We tried running FPL without overflow checks and with a fixed precision level on the subset of test cases that can be run at that precision. For example, running FPL with only 16-bit integers and no overflow checks on the corresponding subset results in a speedup of only 1.08x (Figure 14) over FPL with transprecision. Since running with unchecked 16-bit integers provides a roofline on the possible performance on the subset of test cases that can be run using 16-bit integers, this indicates that FPL is very close to the roofline, and that the overhead of transprecision computing is quite low.

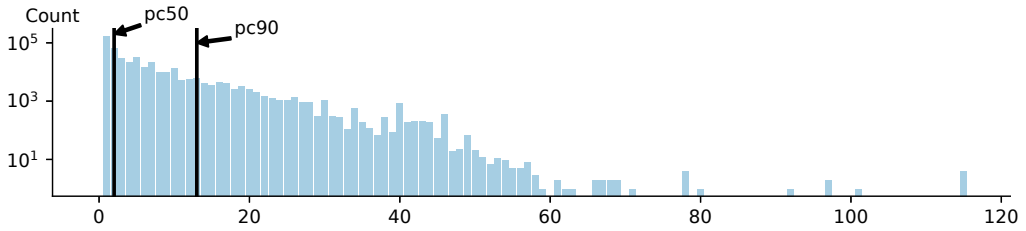


Fig. 15. The distribution of the number of pivots performed by our implementation of the Simplex algorithm when FPL is run on our benchmark suite. We find that the maximum number of pivots performed on any single linear program is 115, which is quite low. As such, our baseline implementation of the Simplex algorithm suffices to solve the linear programs arising in polyhedral compilation.

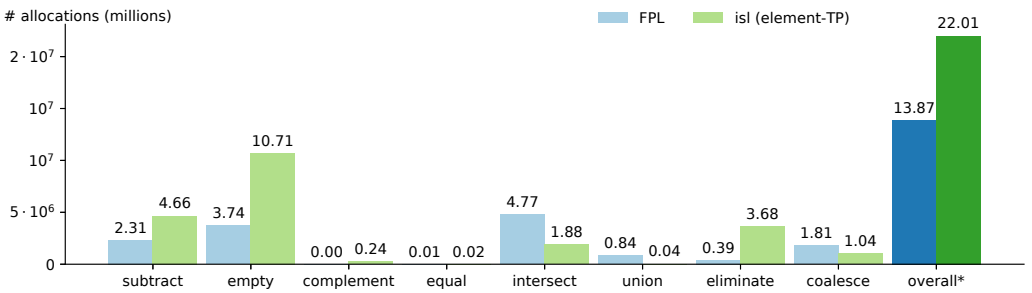


Fig. 16. We find that FPL makes 40% fewer calls to memory allocation primitives than isl, demonstrating the advantage of our library design.

Number of pivots. We investigated the number of pivot operations required by Simplex to solve the linear programs that arise when FPL is run on our benchmark. King et al. [2014] explored a hybrid approach for leveraging linear programming in SMT solvers, where they used an internal exact solver for small cases and called out to an external commercial solver for larger cases that took more than 200 pivots to solve. Such solvers use complex pivoting rules and other optimizations, enabling them to tackle more challenging cases. It is therefore interesting to see how many pivots are required to solve a typical linear program that arises in our use case; if some cases require a large number of pivots, then it may be useful to call out to such a solver from FPL.

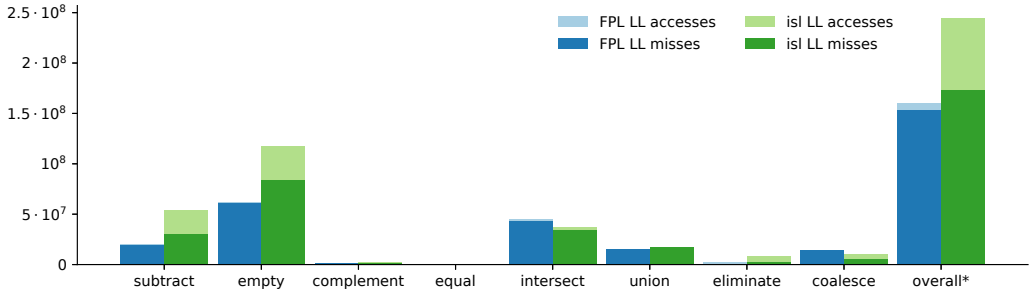


Fig. 17. Simulated cache performance under Valgrind in a two-level cache hierarchy. We find that FPL accesses the last level cache 34% fewer times than isl. Additionally, FPL has 11% fewer last level cache *misses*, and therefore has 11% fewer DRAM accesses.

Our implementation of the Simplex algorithm supports dynamically adding constraints as well as backtracking. We consider that whenever we backtrack and add new constraints, we are solving a new linear program. We find that the maximum number of pivots needed to solve any linear program is only 115 (Figure 15), so our straightforward baseline implementation of the Simplex algorithm is likely sufficient for our use case.

Size of the library. We looked into the effect of our transprecision approach on binary size. Since our library is header-only, the size of our benchmark runner binary's object file provides a good indication of the size of the library. The size of the binary object itself is 1870 KiB when using full transprecision as compared to 570 KiB when using only GMP. The size of the object file plus the dependencies for GMP and overflow checks is 3.47 MiB when using full transprecision and 1.95 MiB when using only GMP. Compared to the size of e.g. the MLIR optimizer `mlir-opt`, which is of size 62 MiB, the size of the library is quite minimal.

Memory behavior. We investigate the benefits of our library design as compared to isl by comparing the number of calls we make to memory allocation primitives. We find that FPL makes 40% fewer memory allocation calls than isl (Figure 16). We also tried running FPL and isl under Valgrind [Nethercote and Seward 2007] to investigate cache behavior under a simulated two-level cache hierarchy. The cache configuration resembles that of the i7-1065G7 CPU mentioned earlier. The configuration has an 8-way associative, 32KiB level 1 instruction cache; a 12-way associative, 48KiB level 1 data cache; and a 16-way associative 8MiB last level cache. We find that FPL accesses the last level cache 34% fewer times than isl, and accesses DRAM 11% fewer times (Figure 17). These results demonstrate the advantages of our library design in terms of memory behavior.

7 RELATED WORK

Many of the algorithms used in this work are well-established in the domain of mathematical optimization. Our library includes an implementation of the Generalized Basis Reduction (GBR) algorithm of Lovász and Scarf [1992], the Parametric Integer Programming (PIP) algorithm of Feautrier [1988], and an implementation of the Simplex algorithm based on the Simplex module of Simplify [Detlefs et al. 2005]. Simplify is a system for automatic theorem proving that supports linear programming and limited heuristics for integer linear programming. Its Simplex module supports linear programming with incremental computation of results while adding and removing constraints, a crucial building block in our implementation of GBR and PIP. The GBR algorithm can solve integer linear programming problems having a fixed number of variables in polynomial

time. We use this for emptiness checks on basic sets, as the two problems are equivalent. The PIP algorithm was developed to support sets with symbolic variables, a requirement arising in program analysis. We use it to eliminate existential variables (Section 4.1.3). The big M method used to support negative-valued variables in PIP is also well-known in the linear programming literature [Bazaraa and Jarvis 1977].

Several math libraries that reason about rational polyhedra, integer polyhedra, and Presburger sets have been built earlier on top of these and similar algorithms. The abstract interpretation community has been developing a range of libraries that operate on convex rational polyhedra and even less generic objects. These libraries include Apron [Jeannet and Miné 2009], PPL [Bagnara et al. 2008], and Elina [Singh et al. 2015]. The Omega project [Kelly et al. 1996] was the first library dedicated to polyhedral compilation. It contributed the Omega check, an integer emptiness algorithm. Omega uses a dual representation of vertex and integer polytopes to represent Presburger sets. Today, Omega is unfortunately unmaintained. Polylib [Loechner 1999] initially allowed reasoning about non-convex sets of rational polyhedra and over time gained support for some integer algorithms, also using a dual representation of vertex and constraint polytopes. Polylib 5.22.5 provides macros that allow users to change the element type of the polylib data structures at compile time and it also has a hand-crafted exception system that Polylib uses to track overflows and free data structures in case an exception occurs, but Polylib does neither use SIMDization nor can it automatically transition from low-precision to higher-precision types. The R-stream compiler [Meister et al. 2011] provides Jolylib, a polyhedral constraint library implemented in Java, but to our understanding, the details of this library have not been publicly discussed. The integer set library (isl) [Verdoolaege 2010] is today the state-of-the-art library for Presburger arithmetic and provides support for full Presburger arithmetic, including existential constraints. The isl project was the first to recognize that a purely constraint-based internal representation is well-suited for the kinds of Presburger sets that arise during program analysis. Also, it was the first to combine algorithms such as GBR, PIP, and Simplify to enable effective reasoning about existential dimensions in Presburger sets, to the best of our knowledge. isl introduced the coalescing algorithm [Verdoolaege 2015] and was our source for the subtraction algorithm as well as the usage of the big M method to support negative variables in PIP. The design of our library follows the algorithmic structure of isl and many of the data structures and design choices we used follow isl – some were originally crafted for isl while others already appeared in earlier libraries such as Omega or Polylib. By documenting the full set of algorithms used in FPL together with a performance analysis of potential design choices, we aim to facilitate future contributions to FPL. The isl library initially only supported GMP as an integer type. Grosser et al. [2020] later added element-wise transprecision computing across the full library and presented a prototype that exploited a simplex solver optimized with matrix-wise transprecision computing. In contrast, FPL uses library-level transprecision computing across the full Presburger library. Finally, the verified polyhedral library (VPL) [Fouilhe 2015] is a formally verified implementation of a library for polyhedral arithmetic in OCaml that is similar to isl and was developed around the same time. It too uses a constraint-only representation. However, it does not support reasoning about integrality or existential dimensions.

8 CONCLUSION

We presented FPL, a new library for Presburger arithmetic built from the ground up with support for transprecision computing at the library level. We showed how FPL's modern design results in a 40% reduction in calls to memory allocation primitives. For the first time, we provided a complete documentation and performance analysis of the algorithmic foundations underlying a Presburger library. We evaluated our library against a newly developed benchmark suite characterizing the set operations and inputs that arise in real-world polyhedral compilation, finding that our library

shows an aggregate 5.4x speedup in terms of total runtime over isl running with GMP and a speedup of 3.6x over isl enhanced with an element-level transprecision optimization. We expect that the availability of a well-documented and fast Presburger library will accelerate the adoption of polyhedral compilation techniques in production compilers.

9 DATA AVAILABILITY STATEMENT

We have made available a replication package [Pitchanathan et al. 2021] that includes a Docker image with the necessary toolchains, data, sources and scripts to reproduce the main results from our evaluation (Section 6). The package also includes our benchmark for Presburger arithmetic as used in polyhedral compilation.

ACKNOWLEDGMENTS

This work has been supported by ARM Ltd. and Xilinx Inc., in the context of Polly Labs. We would also like to thank Kunwar Shaanjeet Singh Grover for contributing additional enhancements to the library.

REFERENCES

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, 193–205. https://dl.acm.org/doi/10.5555/3314872.3314896

Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Sci. Comput. Program.* 72, 1–2 (June 2008), 3–21. https://doi.org/10.1016/j.scico.2007.08.001

M. S Bazaraa and J. J. Jarvis. 1977. *Linear Programming and Network Flows.* John Wiley & Sons, Ltd.

Uday Bondhugula and J. Ramanujam. 2007. *Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer.* Technical Report.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

Paul Feautrier. 1988. Parametric integer programming. *RAIRO-Operations Research* 22, 3 (1988), 243–268. https://doi.org/10.1051/ro/1988220302431

Alexis Fouilhe. 2015. *Revisiting the abstract domain of polyhedra : constraints-only representation and formal proof.* Theses. Université Grenoble Alpes. https://tel.archives-ouvertes.fr/tel-01286086

Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly – Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012). https://doi.org/10.1142/S0129626412500107

Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC Transparent Compilation to Heterogeneous Hardware. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/2925426.2926286

Tobias Grosser, Theodoros Theodoridis, Maximilian Falkenstein, Arjun Pitchanathan, Michael Kruse, Manuel Rigger, Zhendong Su, and Torsten Hoefler. 2020. Fast Linear Programming through Transprecision Computing on Small and Sparse Data. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 195 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428263

Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A Fast Analytical Model of Fully Associative Caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 816–829. https://doi.org/10.1145/3314221.3314606

Christoph Haase. 2018. A Survival Guide to Presburger Arithmetic. *ACM SIGLOG News* 5, 3 (July 2018), 67–82. https://doi.org/10.1145/3242953.3242964

Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The Omega calculator and library, version 1.1. 0. *College Park, MD* 20742 (1996), 18.

Tim King, Clark Barrett, and Cesare Tinelli. 2014. Leveraging linear and mixed integer programming for SMT. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. 139–146. https://doi.org/10.1109/FMCAD.2014.6987606

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

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

Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra. (1999).

László Lovász and Herbert E. Scarf. 1992. The Generalized Basis Reduction Algorithm. *Mathematics of Operations Research* 17, 3 (1992), 751–764. http://www.jstor.org/stable/3689761

Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. *R-Stream Compiler.* Springer US, Boston, MA, 1756–1765. https://doi.org/10.1007/978-0-387-09766-4_515

Kedar S. Namjoshi and Nimit Singhania. 2016. Loopy: Programmable and Formally Verified Loop Transformations. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science)*, Xavier Rival (Ed.), Vol. 9837. Springer, 383–402. https://doi.org/10.1007/978-3-662-53413-7_19

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2020. Automated Derivation of Parametric Data Movement Lower Bounds for Affine Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 808–822. https://doi.org/10.1145/3385412.3385989

Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser. 2021. Replication Package for Article: FPL: Fast Presburger Arithmetic through Transprecision. (2021). https://doi.org/10.1145/3462302

Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. (2012).

Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-Based Data Reuse Optimization for Configurable Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 29–38. https://doi.org/10.1145/2435264.2435273

Alexander Schrijver. 1986. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., USA.

Ron Shamir. 1987. The Efficiency of the Simplex Method: A Survey. *Management Science* 33, 3 (1987), 301–334. https://doi.org/10.1287/mnsc.33.3.301

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2015. Making Numerical Program Analysis Fast. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 303–313. https://doi.org/10.1145/2737924.2738000

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv e-prints*, Article arXiv:1802.04730 (Feb. 2018), arXiv:1802.04730 pages. arXiv:cs.PL/1802.04730

Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *ICMS*, Vol. 6327. 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

Sven Verdoolaege. 2015. Integer set coalescing. In *International Workshop on Polyhedral Compilation Techniques, Date: 2015/01/19-2015/01/19, Location: Amsterdam, The Netherlands.*

Sven Verdoolaege. 2016. Presburger formulas and polyhedral compilation. (2016).

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713