Fast Algorithms for Octagon Abstract Domain

Fast Algorithms for Octagon Abstract Domain

Gagandeep Singh

Master Thesis
April 2014

Supervisors:
Prof. Dr. Martin Vechev, Prof. Dr. Markus Püschel

Abstract

Numerical abstract domains are used to discover useful numerical properties about program code such as invariants and absence of runtime errors. There are a number of numerical domains with varying degrees of precision and cost. The Octagon domain is one of the most popular numerical domains which aims to strike a balance between precision and cost. However, the cost of the Octagon domain can still be an issue in practice, especially if the program uses many variables. This thesis develops new, optimized algorithms for some of the most expensive and frequently used operations of the Octagon domain leading to significant speed-ups over existing implementations.

Acknowledgment

I would like to thank Prof. Dr. Martin Vechev and Prof. Dr. Markus Püschel for supervising this project. Thanks to Prof. Dr. Vechev I have learned a lot about static analysis that was not very familiar to me before joining ETH Zurich. I have learned a lot of optimization techniques from Prof. Dr. Püschel that helped me throughout this project. Furthermore, I would like to thank members of the computer science department who directly or indirectly helped me in this project. In the end, I would like to thank my parents, sister and my late grandmother for all the sacrifices they have made over the years.

Contents

List of Figures

List of Tables

1 Introduction

The goal of static program analysis is to automatically discover useful properties about programs. These properties can then be used to prove the presence or absence of runtime errors such as null pointer errors, buffer overflows and concurrency errors. There are a number of abstract domains that are commonly employed for such analysis.

When designing abstract domains, there is always a tradeoff between precision of the domain and its cost. Consider the set of points shown in Figure 1.1(a). These points represent the set of concrete values that two variables x and y can take at any given point in some program.

Interval Domain The Interval domain is a non-relational numerical abstract domain. It stores the lower and upper bounds for variables in the program. It is very cheap and has linear asymptotic space and time complexity with respect to the number of program variables. The abstraction for the set of points in Figure 1.1(a) under the Interval domain is shown in Figure 1.1(b). This abstraction is quite imprecise as it cannot capture relationships between variables. For instance, this domain cannot capture the fact that $x < y$.

Polyhedra Domain The Polyhedra[13, 8] domain is a relational abstract numerical domain. It maintains linear relationship between program variables of the form $\sum_i a_i v_i \leq c$. It is very expensive and has exponential asymptotic space and time complexity with respect to the number of program variables. The abstraction for the set of points in Figure 1.1(a) under the Polyhedra domain is shown in figure 1.1(c).

Octagon Domain The Octagon abstract domain[26] is a weakly relational numerical domain sitting between the non-relational interval abstract domain and the relational Polyhedral domain (in terms of the cost vs precision tradeoffs). It has quadratic asymptotic space and cubic asymptotic time complexity with respect to the number of program variables. The abstraction for the set of points in Figure 1.1(a) under the Octagon domain is shown in Figure 1.1(d).

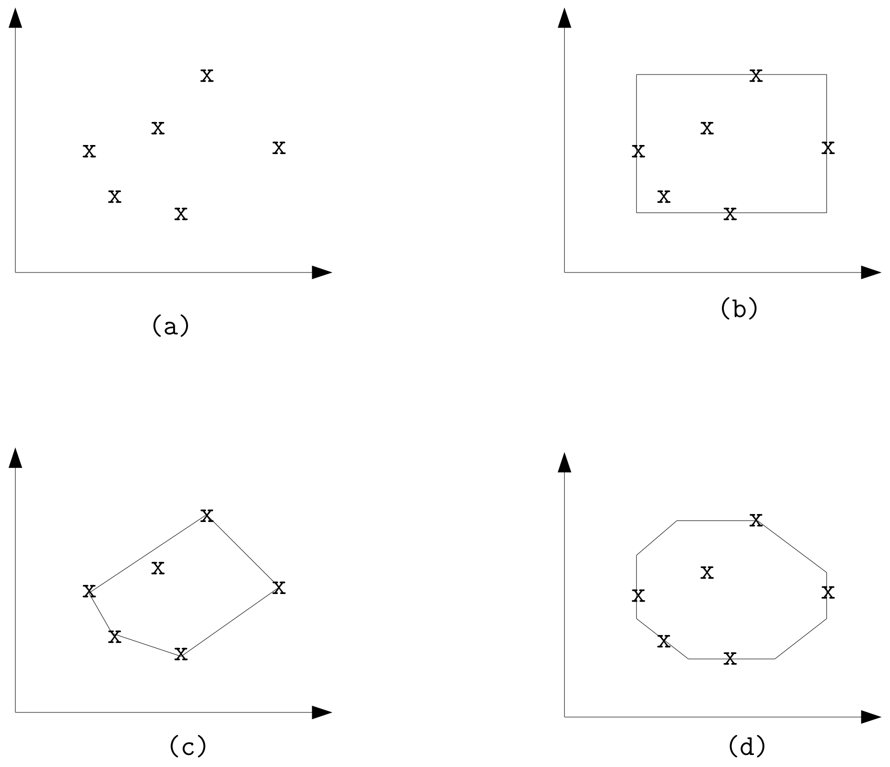

Figure 1.1.: (a) A set of points (b) its abstraction in Interval (c) its abstraction in Polyhedra (d) its abstraction in Octagon

1.1. Key Challenges

Although the Octagon abstract domain is faster than Polyhedra, it still has cubic asymptotic time complexity which can be prohibitively slow when analyzing real-world programs. Further, current implementations of the Octagon domain do not heavily focus on optimizing its performance. Most of the work for improving the runtime of the Octagon analysis is focused on approximating expensive operators thus resulting in precision loss.

The cubic time complexity occurs due to the *closure* operation, a particular core operation used by many Octagon operators. In this thesis, we focus on improving the performance of the closure operation by using memory optimizations, vector hardware instructions [3] and exploiting sparsity and structure of octagons. A key objective of our work is to devise a faster Octagon analysis without losing any precision. To achieve that, we add a linear amount of space.

1.2. Contributions

Our work makes the following contributions:

- A novel algorithm for computing *closure*. Our algorithm brings down the number of operations to half as compared to existing algorithms. We also provide a vector implementation of this algorithm using Intel's AVX intrinsics.
- A novel sparse algorithm for closure which exploits sparsity in matrices.
- A novel algorithm for computing incremental closure which brings down the number of operations to half compared to existing algorithms. We also provide a vector implementation of this algorithm using Intel's AVX intrinsics.
- A novel sparse algorithm for incremental closure which exploits sparsity in matrices.
- A vector implementation of common Octagon operators using Intel's AVX intrinsics.
- A sparse and dense library implementation of the Octagon domain which provides massive speedup over popular Octagon implementations.

1.3. Structure of this Document

This thesis is structured as follows. Chapter 2 describes background on static analysis and the Octagon domain as well as existing implementations of the Octagon domain. Chapter 3 describes the closure operator which is a frequently used but expensive operator of the Octagon domain and presents our novel algorithms which speed-up the closure operation. Chapter 4 describes the remaining operators of the Octagon domain and our proposed optimizations. Chapter 5 discusses the design and implementation of our sparse and dense libraries for the Octagon analysis. We also compare the performance of our libraries against a popular static analysis library on a number of benchmarks. Finally, Chapter 6 discusses future work.

1.4. Conventions

Throughout this document, we use the following conventions:

- Strong closure is referred to as closure.
- Some Octagon domain operators behave differently for integers and reals. In this thesis, we deal only with reals.
- Whenever we mention complexity, we refer to asymptotic complexity that is calculated with respect to number of program variables.
- There is some distinction between transformers and operators. For the sake of simplicity, we refer to both transformers and operators as operators.

2

Background

This chapter provides the necessary background on static analysis: we briefly describe the basics of abstract interpretation, numerical domains and the Octagon abstract domain.

2.1. Abstract Interpretation

Abstract interpretation[12] computes an over approximation of the program behaviors. It works via two domains: the concrete domain C and the abstract domain A. An element in C captures the program behaviors whereas elements of the abstract domain A define potential over approximations of the concrete program behaviors in C. Thus, some behaviors in the abstract may never occur in the concrete, but all concrete behaviors will naturally be captured by an over-approximation in A. The concrete and abstract domains are related via two functions:

- An abstraction function α that maps an element x in C to an element $\alpha(x)$ in A. The element $\alpha(x)$ is called an abstraction of x.
- A concretization function γ that maps an element y in A to an element $\gamma(y)$ in C. The element $\gamma(y)$ is called a concretization of y.

These functions usually define a Galois connection as shown in Figure 2.1, meaning that:

- $\alpha : C \to A$ is monotone
- $\gamma : A \to C$ is monotone
- $\forall x \in C, x \sqsubseteq \gamma(\alpha(x))$
- $\forall y \in A, y \sqsupseteq \alpha(\gamma(y))$

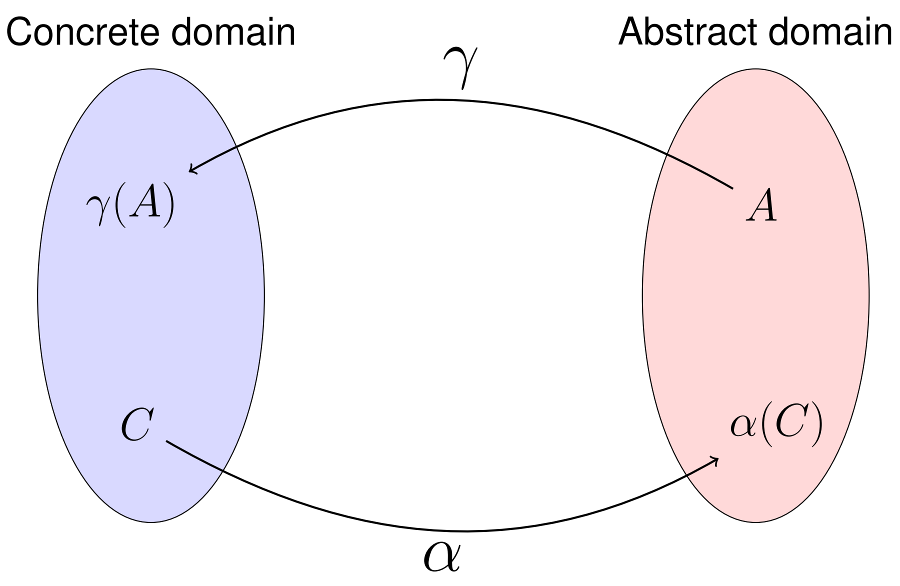

Figure 2.1.: Galois connection between abstract and concrete domains

Abstract and concrete domains typically support the following operators:

- The join $\sqcup$ operator.
- The meet $\sqcap$ operator.
- The ordering $\sqsubseteq$ operator.
- The widening $\bigtriangledown$ operator.
- The transformer function which iterates over the domain and models the effect of program statements.

To ensure soundness, operators in the abstract domain over-approximate the effect of applying the concrete operators. These operators define monotone functions over the domain. The concrete semantics of a program are defined by a program invariant X such that after applying a concrete semantic function f, we have that $X = f(X)$. The value of X is usually not computable, and therefore we use the abstract semantics and compute an (abstract) invariant $Y = \alpha(X)$ such that after applying the abstract semantic function g we have that $Y = g(Y)$. Since g is monotone and α and γ form a Galois connection, we have $X \sqsubseteq \gamma(g(\alpha(X)))$ and the abstraction is sound.

2.2. Numerical Domains

In this thesis, we focus squarely on numerical abstract domains. A numerical domain abstracts sets of numerical values (of program variables). These domains are concerned with finding useful numerical properties relating numerical program variables. These properties can then be used to prove presence or absence of common numerical runtime errors such as division by zero, buffer overflow and overflow of machine integers.

The numerical domain can be relational or non-relational. A relational abstract domain maintain relationship between program variables whereas a non-relational domain does not. An example of a relational abstract domain is Polyhedra whereas the Interval abstract domain is an example of a non-relational domain.

```
$x \leftarrow 5$
$y \leftarrow 0$
while $x > 0$ do
    invariant: $x + y \leq 5$
    $x \leftarrow x - 1$
    $y \leftarrow y + 1$
assert: $y \leq 5$
```

Figure 2.2.: Assertion with Octagons

```
$x \leftarrow 5$
$y \leftarrow 0$
$z \leftarrow 0$
while $x > 0$ do
    invariant: $x + y + z \leq 10$
    $x \leftarrow x - 1$
    $y \leftarrow y + 1$
    $z \leftarrow z + 1$
assert: $y + z \leq 10$
```

Figure 2.3.: Assertion with Polyhedra

Consider the code fragment shown in Figure 2.2. In order to prove the non relational invariant $y \leq 5$ at the end of the loop, we need to first prove the relational loop invariant $x + y \leq 5$ and then combine that with the loop exit condition $x = 0$. This invariant can be established with the relational polyhedron domain but not with the non-relational interval domain.

2.3. Octagon Abstract Domain

The Octagon abstract domain is a weakly relational domain that supports a more limited number of relations between program variables than a polyhedron. It encodes binary constraints between program variables of the form $c_i x_i + c_j x_j \leq c$ where x_i and x_j are program variables, $c_i, c_j \in [-1, 0, 1]$ and $c \in \mathbb{R} \cup \{\infty\}$. Since coefficients can be either $-1, 0$ or 1 the number of inequalities between any two variables is bounded. The data structure used to encode relations between program variables are difference bound matrices (DBMs)[25]. Let $\{v_1, v_2 \dots v_n\}$ be the set of n variables for a program. Then, the Octagon domain introduces two variables v_i^+ and v_i^- for each program variable v_i. Thus, a DBM would be of size $2n \times 2n$. An element $m_{i,j} = c$ in DBM encodes the relationship of the form $v_j - v_i \leq c$. DBM's satisfy a coherence property i.e., that the elements $m_{i,j}$ and $m_{j\oplus 1, i\oplus 1}$ encode the same constraint.

The invariant $y \leq 5$ for the code fragment in Figure 2.2 can be proved using the Octagon domain as the loop invariant $x + y \leq 5$ can be encoded as an octagonal constraint.

Consider the code fragment shown in Figure 2.3. In order to prove the relational invariant $y + z \leq 10$ at the end of the loop, we need to first prove the non octagonal loop invariant $x + y + z \leq 10$ and then combine that with the loop exit condition $x = 0$. This invariant can be established with a more precise polyhedron domain but not with the Octagon domain.

The main advantage of Octagon domain over a polyhedron is that it requires lower time complexity. That is, it provides reasonable precision with polynomial complexity: it has quadratic space and cubic time complexity compared with exponential for polyhedron.

2.4. Related Work

Figure 2.4 shows the work flow between different components of a typical static analyzer. The input program is first parsed by the front end to build semantic equations which model the effect of program statements like assignment, guards etc. The equations are then passed to an abstract domain that provides the representation for abstract element and operators to operate on it. The equations are then solved by a solver which uses the abstract domain towards a fixpoint computation.

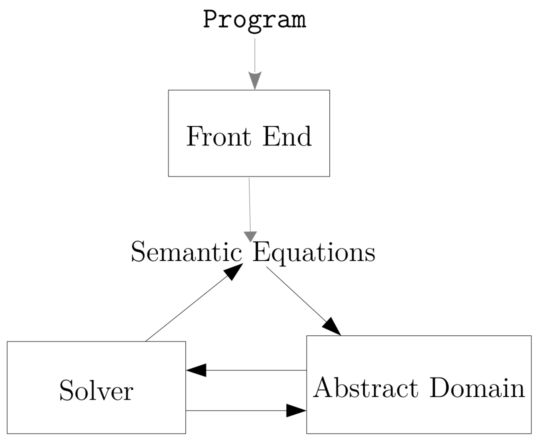

Figure 2.4.: Work flow between different components of a static analyzer

APRON[19] is a popular static analysis library that provides common interface for various numerical abstract domains such as Interval, Octagon, Convex Polyhedra, Linear equalities and Linear Congruences. The library is written in C but can also be accessed from C++, Java and OCaml. It provides a manager class which can be instantiated to use any of the numerical domains. All of the domains support the same operations such as join, meet, widening etc. The abstract domain can be accessed using either double precision, rational numbers or integers.

The Parma Polyhedra Library(PPL)[4] also provides support for the Octagon domain. The library is written in C++ but can also be accessed from C, Java, OCaml and Prolog.

Astrée[23] is a static analyzer which uses Octagons for proving absence of runtime properties. It is widely employed in industry and was used by Airbus to prove absence of runtime errors for the Airbus A380. The work of [6] provides an implementation of the Octagon domain on Graphical Processing Units (GPUs). Since the data structure commonly used for representing octagons is a matrix, it can be easily implemented in a GPU. They implement joins, meets, assignments, guard checks, closure, widening, and the fixpoint checking on the GPU.

An alternative, bucketing approach to speeding up the Octagon analysis has been proposed independently in [7] and [32] (the idea is to handle large octagons). They divide a large octagon of n variables into a set of k smaller octagons called buckets with each set having at most n/k variables. The buckets may share some variables that are called pivots. The analysis is run on each set which introduces some precision loss. This approach also needs to address the problem of partitioning the program variables into buckets. Existing heuristics usually use the structure of the source program to perform this partitioning.

3

Closure Algorithms

The *closure* operator is the one of the most frequently used operators in the Octagon domain. It is used to make implicit constraints between program variables explicit. It is used as a pre-processing step to increase precision of various operators such as join, equality and inclusion testing. Its cubic time complexity makes it the most expensive operator of the Octagon domain. For some domains [22], attempts have been made to avoid computing closure to avoid cubic complexity. In this work, we try to optimize closure for runtime performance by taking advantage of memory optimizations, vector hardware instructions, and structure and sparsity of DBMs which are often encountered in analysis of real world programs.

As defined originally by Miné [26], the DBM matrix *m* obtained after applying closure should satisfy the following properties:

- *m* should be coherent i.e., $\forall i, j, m_{i,j} \leftarrow m_{\bar{j},\bar{i}}$
- *m* is closed i.e., $\forall i,\ m_{i,i} \leftarrow 0$ and $\forall i, j, k,\ m_{i,j} \leq m_{i,k} + m_{k,j}$
- $\forall i, j,\ m_{i,j} \leq (m_{i,\bar{i}} + m_{\bar{j},j})/2$

We next discuss the algorithms for computing closure, both existing as well as our novel algorithms.

3.1. Miné's Algorithm

Algorithm 1 shows the pseudo code for computing closure as originally proposed by Miné [26]. The algorithm consists of two parts: The first part is a modified version of the classic Floyd-Warshall all pairs shortest path algorithm [15] [17] [33] and ensures that properties 1 and 2 are satisfied. The second part is a strengthening step and ensures that property 3 is satisfied. Notice that the strengthening is applied linear number of times. The algorithm returns the Bottom element (see appendix A) in the presence of negative cycle. Next, we discuss the optimizations for this algorithm.

Algorithm 1 Miné's Closure

```
1.  function MINÉ'S_CLOSURE(m,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $dim \leftarrow$ number of variables in program
5.      $n \leftarrow 2 * dim$
6.      for $k \leftarrow 0$ to $dim$ do
7.          $p \leftarrow 2k, q \leftarrow 2k+1$
8.          for $i \leftarrow 0$ to $n$ do
9.              for $j \leftarrow 0$ to $n$ do
10.                 $m_{i,j} \leftarrow min(m_{i,j}, m_{i,p} + m_{p,j}, m_{i,q} + m_{q,j}, m_{i,p} + m_{p,q} + m_{q,j}, m_{i,q} + m_{q,p} + m_{p,j})$
11.         $strengthening(m, dim)$
12.     return $consistent(m, dim)$
```

Algorithm 2 Strengthening

```
1.  function STRENGTHENING(m,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $dim \leftarrow$ number of variables in program
5.      $n \leftarrow 2 * dim$
6.      for $i \leftarrow 0$ to $n$ do
7.          for $j \leftarrow 0$ to $n$ do
8.              $m_{i,j} \leftarrow min(m_{i,j}, (m_{i,i\oplus 1} + m_{j\oplus 1,j})/2)$
```

Algorithm 3 Consistent

```
1.  function CONSISTENT(m,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $dim \leftarrow$ number of variables in program
5.      $n \leftarrow 2 * dim$
6.      for $i \leftarrow 0$ to $n$ do
7.          if $m_{i,i} < 0$ then
8.              return $Bottom$
9.      return $m$
```

3.1.1. Scalar Optimizations

A naive implementation of Algorithm 1 will have the four comparisons at lines 10 nested so that the outer min operations have to wait for the result from the inner ones. However, the comparisons are independent of each other and can be implemented in parallel to increase instruction level parallelism.

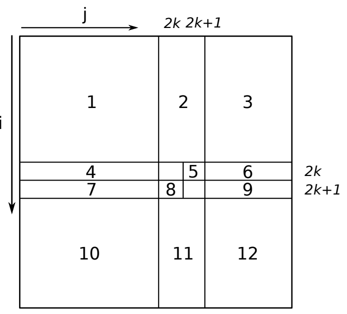

Figure 3.1.: Block view of closure computation

Algorithm 4 Miné’s Closure Scalar Optimized

```
function MINÉ’S_CLOSURE_SCALAR(m,a,b,c,d,e,f,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $a \leftarrow$ array to store old 2k-th column values
        $b \leftarrow$ array to store old (2k+1)-th column values
        $c \leftarrow$ array to store updated 2k-th column values
        $d \leftarrow$ array to store updated (2k+1)-th column values
        $e \leftarrow$ array to store old 2k-th row values
        $f \leftarrow$ array to store old (2k+1)-th row values
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $k \leftarrow 0$ to $dim$ do
        $p \leftarrow 2k, q \leftarrow 2k+1$
        $t_1 \leftarrow m_{p,q}$
        $t_2 \leftarrow m_{q,p}$
        $m_{p,q} \leftarrow min(m_{p,q}, min(m_{p,q} + m_{q,p} + m_{p,q}))$
        $m_{q,p} \leftarrow min(m_{q,p}, min(m_{q,p} + m_{p,q} + m_{q,p}))$
        for $i \leftarrow 0$ to $p$ do
            $a_i \leftarrow m_{i,p}, b_i \leftarrow m_{i,q}$
            $m_{i,p} \leftarrow min(m_{i,p}, min(m_{i,q} + t_2, m_{i,p} + t_1 + t_2))$
            $m_{i,q} \leftarrow min(m_{i,q}, min(m_{i,p} + t_1, m_{i,q} + t_2 + t_1))$
            $c_i \leftarrow m_{i,p}, d_i \leftarrow m_{i,q}$
        for $i \leftarrow q+1$ to $n$ do
            $a_i \leftarrow m_{i,p}, b_i \leftarrow m_{i,q}$
            $m_{i,p} \leftarrow min(m_{i,p}, min(m_{i,q} + m_{q,p}, m_{i,p} + m_{p,q} + m_{q,p}))$
            $m_{i,q} \leftarrow min(m_{i,q}, min(m_{i,p} + m_{p,q}, m_{i,q} + m_{q,p} + m_{p,q}))$
            $c_i \leftarrow m_{i,p}, d_i \leftarrow m_{i,q}$
        for $j \leftarrow 0$ to $p$ do
            $e_j \leftarrow m_{p,j}$
            $m_{p,j} \leftarrow min(m_{p,j}, min(t_1 + m_{q,j}, t_1 + t_2 + m_{p,j}))$
        for $j \leftarrow q+1$ to $n$ do
            $e_j \leftarrow m_{p,j}$
            $m_{p,j} \leftarrow min(m_{p,j}, min(m_{p,q} + m_{q,j}, m_{p,q} + t_2 + m_{p,j}))$
        for $j \leftarrow 0$ to $p$ do
            $f_j \leftarrow m_{q,j}$
            $m_{q,j} \leftarrow min(m_{q,j}, min(t_2 + m_{p,j}, t_2 + m_{p,q} + m_{q,j}))$
        for $j \leftarrow q+1$ to $n$ do
            $f_j \leftarrow m_{q,j}$
            $m_{q,j} \leftarrow min(m_{q,j}, min(m_{q,p} + m_{p,j}, m_{q,p} + m_{p,q} + m_{q,j}))$
        for $i \leftarrow 0$ to $p$ do
            $ik \leftarrow a_i, ikk \leftarrow b_i$
            for $j \leftarrow 0$ to $p$ do
                $kj \leftarrow e_j, kkj \leftarrow f_j$
                $op_1 \leftarrow min(ik + kj, ikk + kkj), op_2 \leftarrow min(ik + t_1 + kkj, ikk + t_2 + kj)$
                $m_{i,j} \leftarrow min(m_{i,j}, min(op_1, op_2))$
            $ik \leftarrow c_i, ikk \leftarrow d_i$
            for $j \leftarrow q+1$ to $n$ do
                $kj \leftarrow e_j, kkj \leftarrow f_j$
                $op_1 \leftarrow min(ik + kj, ikk + kkj), op_2 \leftarrow min(ik + t_1 + kkj, ikk + t_2 + kj)$
                $m_{i,j} \leftarrow min(m_{i,j}, min(op_1, op_2))$
        for $i \leftarrow q+1$ to $n$ do
            $ik \leftarrow a_i, ikk \leftarrow b_i$
            for $j \leftarrow 0$ to $p$ do
                $kj \leftarrow m_{p,j}, kkj \leftarrow m_{q,j}$
                $op_1 \leftarrow min(ik + kj, ikk + kkj), op_2 \leftarrow min(ik + m_{p,q} + kkj, ikk + m_{q,p} + kj)$
                $m_{i,j} \leftarrow min(m_{i,j}, min(op_1, op_2))$
            $ik \leftarrow c_i, ikk \leftarrow d_i$
            for $j \leftarrow q+1$ to $n$ do
                $kj \leftarrow m_{p,j}, kkj \leftarrow m_{q,j}$
                $op_1 \leftarrow min(ik + kj, ikk + kkj), op_2 \leftarrow min(ik + m_{p,q} + kkj, ikk + m_{q,p} + kj)$
                $m_{i,j} \leftarrow min(m_{i,j}, min(op_1, op_2))$
        return $strengthening\_scalar(m, a, dim)$
```

Because for a given iteration k of the outermost loop, only the elements in the $2k$-th and the $(2k+1)$-th rows and columns are accessed as operands, the memory access performance can be increased by storing the two columns in an array to avoid page faults and TLB misses. However, care must be taken as the values in the two rows and columns change during the iteration and thus some elements require older values whereas others require updated ones. We divide the matrix as shown in Figure 3.1 into blocks with each block requiring different versions of operands. The numbering shows the order in which blocks are computed by the original computation. For example, elements in block 1 require $2k$ and $(2k+1)$-th row and column computed in $(k-1)$-th iteration whereas in contrast elements in block 12 require an updated $2k$ and $(2k+1)$-th row and column computed in k-th iteration.

Algorithm 4 shows the pseudo code for the optimized Miné's algorithm. We first compute the elements $m_{2k,2k+1}$ and $m_{2k+1,2k}$ corresponding to blocks 5 and 8 respectively. Some elements will require old values of these elements so we store the old values at lines 14 and 15 . We then perform k-th iteration on elements in $2k$ and $(2k+1)$-th rows and columns. We store old values for both row and column in an array. The updated column values are also stored in an array. Storing column values in an array improves memory access performance. We also reduce opcount for the computation of elements in $2k$ and $(2k+1)$-th row and column by removing redundant terms. We then compute elements in blocks 1,3,10 and 12 using the old and updated values of operands.

The strengthening as shown in Algorithm 2 creates many TLB misses for large matrices as operands for the min operation at line 8 are diagonal elements. It can be easily shown that the diagonal elements do not change during strengthening and can therefore be stored in an array which improves memory performance. The pseudo code for the optimized strengthening is shown in Algorithm 5.

Algorithm 5 Strengthening Scalar Optimized

```
function STRENGTHENING_SCALAR(m,t, dim)
    Parameters:
        $m \leftarrow$ input matrix
        $t \leftarrow$ array to store diagonal elements
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $i \leftarrow 0$ to $n$ do
        $t_i \leftarrow m_{i\oplus 1,i}$
    for $i \leftarrow 0$ to $n$ do
        $ii \leftarrow t_{i\oplus 1}$
        for $j \leftarrow 0$ to $n$ do
            $jj \leftarrow t_j$
            $m_{i,j} \leftarrow min(m_{i,j}, (ii + jj)/2)$
    for $i \leftarrow 0$ to $n$ do
        if $m_{i,i} < 0$ then
            return $Bottom$
    return $m$
```

Algorithm 6 Miné’s Closure AVX

```
function MINÉ'S_CLOSURE_AVX(m,a,b,c,d,e,f,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $a \leftarrow$ array to store old 2k-th column values
        $b \leftarrow$ array to store old (2k+1)-th column values
        $c \leftarrow$ array to store updated 2k-th column values
        $d \leftarrow$ array to store updated (2k+1)-th column values
        $e \leftarrow$ array to store old 2k-th row values
        $f \leftarrow$ array to store old (2k+1)-th row values
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $k \leftarrow 0$ to $dim$ do
        $p \leftarrow 2k, q \leftarrow 2k+1, l \leftarrow pad(q+1)$
        $t_1 \leftarrow m_{p,q}, t_2 \leftarrow m_{q,p}$
        $vt_1 \leftarrow avx\_set\_double(t_1), vt_2 \leftarrow avx\_set\_double(t_2)$
        $m_{p,q} \leftarrow min(m_{p,q}, min(m_{p,q} + m_{q,p} + m_{p,q})), m_{q,p} \leftarrow min(m_{q,p}, min(m_{q,p} + m_{p,q} + m_{q,p}))$
        $vt_3 \leftarrow avx\_set\_double(m_{p,q}), vt_4 \leftarrow avx\_set\_double(m_{q,p})$
        for $i \leftarrow 0$ to $p$ do
            $a_i \leftarrow m_{i,p}, b_i \leftarrow m_{i,q}$
            $m_{i,p} \leftarrow min(m_{i,p}, min(m_{i,q} + t_2, m_{i,p} + t_1 + t_2))$
            $m_{i,q} \leftarrow min(m_{i,q}, min(m_{i,p} + t_1, m_{i,q} + t_2 + t_1))$
            $c_i \leftarrow m_{i,p}, d_i \leftarrow m_{i,q}$
        for $i \leftarrow q+1$ to $n$ do
            $a_i \leftarrow m_{i,p}, b_i \leftarrow m_{i,q}$
            $m_{i,p} \leftarrow min(m_{i,p}, min(m_{i,q} + m_{q,p}, m_{i,p} + m_{p,q} + m_{q,p}))$
            $m_{i,q} \leftarrow min(m_{i,q}, min(m_{i,p} + m_{p,q}, m_{i,q} + m_{q,p} + m_{p,q}))$
            $c_i \leftarrow m_{i,p}, d_i \leftarrow m_{i,q}$
        for $j \leftarrow 0$ to $p$ do
            $e_j \leftarrow m_{p,j}$
            $m_{p,j} \leftarrow min(m_{p,j}, min(t_1 + m_{q,j}, t_1 + t_2 + m_{p,j}))$
        for $j \leftarrow q+1$ to $n$ do
            $e_j \leftarrow m_{p,j}$
            $m_{p,j} \leftarrow min(m_{p,j}, min(m_{p,q} + m_{q,j}, m_{p,q} + t_2 + m_{p,j}))$
        for $j \leftarrow 0$ to $p$ do
            $f_j \leftarrow m_{q,j}$
            $m_{q,j} \leftarrow min(m_{q,j}, min(t_2 + m_{p,j}, t_2 + m_{p,q} + m_{q,j}))$
        for $j \leftarrow q+1$ to $n$ do
            $f_j \leftarrow m_{q,j}$
            $m_{q,j} \leftarrow min(m_{q,j}, min(m_{q,p} + m_{p,j}, m_{q,p} + m_{p,q} + m_{q,j}))$
        for $i \leftarrow 0$ to $p/4$ do
            $ik \leftarrow avx\_set\_double(a_i), ikk \leftarrow avx\_set\_double(b_i)$
            for $j \leftarrow 0$ to $p/4$ do
                $ij \leftarrow avx\_load\_double(m_{i,j*4}), kj \leftarrow avx\_load\_double(e_{j*4}), kkj \leftarrow avx\_load\_double(f_{j*4})$
                $op \leftarrow compute\_elem\_avx(ij, ik, ikk, kj, kkj, vt_1, vt_2)$
                $avx\_store\_double(m_{i,j*4}, op)$
            $ik \leftarrow avx\_set\_double(c_i), ikk \leftarrow avx\_set\_double(d_i)$
            for $j \leftarrow l$ to $n/4$ do
                $ij \leftarrow avx\_load\_double(m_{i,j*4}), kj \leftarrow avx\_load\_double(e_{j*4}), kkj \leftarrow avx\_load\_double(f_{j*4})$
                $op \leftarrow compute\_elem\_avx(ij, ik, ikk, kj, kkj, vt_1, vt_2)$
                $avx\_store\_double(m_{i,j*4}, op)$
        for $i \leftarrow l$ to $n/4$ do
            $ik \leftarrow avx\_set\_double(a_i), ikk \leftarrow avx\_set\_double(b_i)$
            for $j \leftarrow 0$ to $p/4$ do
                $ij \leftarrow avx\_load\_double(m_{i,j*4}), kj \leftarrow avx\_load\_double(m_{p,j*4}), kkj \leftarrow avx\_load\_double(m_{q,j*4})$
                $op \leftarrow compute\_elem\_avx(ij, ik, ikk, kj, kkj, vt_3, vt_4)$
                $avx\_store\_double(m_{i,j*4}, op)$
            $ik \leftarrow avx\_set\_double(c_i), ikk \leftarrow avx\_set\_double(d_i)$
            for $j \leftarrow l$ to $n/4$ do
                $ij \leftarrow avx\_load\_double(m_{i,j*4}), kj \leftarrow avx\_load\_double(m_{p,j*4}), kkj \leftarrow avx\_load\_double(m_{q,j*4})$
                $op \leftarrow compute\_elem\_avx(ij, ik, ikk, kj, kkj, vt_3, vt_4)$
                $avx\_store\_double(m_{i,j*4}, op)$
        return $strengthening\_avx(m, a, dim)$
```

3.1.2. Vectorization

Miné’s algorithm can also be vectorized using Intel’s AVX [3] intrinsics. The pseudo code for Miné’s algorithm with AVX is shown in Algorithm 6. We vectorize the computations in blocks 1,3, 10 and 12. The function pad at line 13 returns the smallest integer $l \geq 2k + 2$. If p is not divisible by 4 then we perform scalar computation of remaining $m_{i,j}$ while $j < p$ after the loops at lines 42 and 53. Similarly, we perform scalar computation from $j = 2k + 2$ to $j = l - 1$ before entering the loops at lines 47 and 58. In the case where n is not divisible by 4, we again perform scalar computation of $m_{i,j}$ after loops at lines 47 and 58 while $j < n$. For simplicity, such border cases are omitted from the pseudo code.

Algorithm 7 Compute Element AVX

```
function COMPUTE_ELEM_AVX($ij, ik, ikk, kj, kkj, vt_1, vt_2$)
    $op_1 \leftarrow avx\_add\_double(ik, kj)$
    $op_2 \leftarrow avx\_add\_double(ikk, kkj)$
    $op_3 \leftarrow avx\_min\_double(op_1, op_2)$
    $op_4 \leftarrow avx\_add\_double(ik, vt_1)$
    $op_4 \leftarrow avx\_add\_double(op_4, kkj)$
    $op_5 \leftarrow avx\_add\_double(ikk, vt_2)$
    $op_5 \leftarrow avx\_add\_double(op_5, kj)$
    $op_6 \leftarrow avx\_min\_double(op_4, op_5)$
    $op_7 \leftarrow avx\_min\_double(op_3, op_6)$
    $op_8 \leftarrow avx\_min\_double(ij, op_7)$
    return $op_8$
```

Algorithm 8 Strengthening AVX

```
1.  function STRENGTHENING_AVX(m,t,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $t \leftarrow$ array to store diagonal elements
5.          $dim \leftarrow$ number of variables in program
6.      $n \leftarrow 2 * dim$
7.      for $i \leftarrow 0$ to $n$ do
8.          $t_i \leftarrow m_{i\oplus 1,i}$
9.      for $i \leftarrow 0$ to $n$ do
10.         $ii \leftarrow avx\_set1\_double(t_{i\oplus 1})$
11.         for $j \leftarrow 0$ to $n/4$ do
12.             $jj \leftarrow avx\_load\_double(t_{j*4})$
13.             $op_1 \leftarrow avx\_load\_double(m_{i,j*4})$
14.             $op_2 \leftarrow avx\_add\_double(ii, jj)$
15.             $op_2 \leftarrow avx\_mul\_double(op_2, 0.5)$
16.             $op_3 \leftarrow avx\_min\_double(op_1, op_2)$
17.             $avx\_store\_double(m_{i,j*4}, op_3)$
18.     for $i \leftarrow 0$ to $n$ do
19.         if $m_{i,i} < 0$ then
20.             return $Bottom$
21.     return $m$
```

The strengthening can also be vectorized using AVX. Algorithm 8 shows the pseudo code for the vectorized strengthening. Notice that storing diagonal elements in an array allows us to perform vectorization. Again, we omit border cases when n is not divisible by 4 for the loop at line 11.

3.2. Sparse Closure

The matrices arising from analyzing real world programs often contain a large number of ∞ values. The reason is that for real programs not all variables are related to each other and unrelated variables have no constraint between them. Thus, usually a given variable is related to only a small number of other variables. From property 2 of the Octagon closure operator, it is clear that if either $m_{i,k}$ or $m_{k,j}$ is ∞ then $m_{i,j}$ does not change and thus addition and min operations can be avoided. Thus, it makes sense to consider a sparse representation of the matrix which only stores finite values.

Bagnara et al. [5] show that it is possible to compute Octagon closure by applying Floyd-Warshall followed by a single strengthening step (see appendix A). We now present a sparse closure algorithm based on Floyd-Warshall. There are various sparse representations available in literature, for example compressed sparse row (CSR) [16]. However, using such a representation would incur an extra overhead in the case when the matrix does not remain "sparse enough" as the analysis proceeds and then we would like to switch to a dense representation which would have a faster dense closure algorithm. We thus sacrifice some space in order to improve the speed and create an index array storing locations of finite values.

To reduce the memory overhead caused due to keeping an extra index, we do not create the index for the whole matrix. Instead, we keep the index for elements only in the k-th row and column during the k-th iteration. This reduces extra space overhead to linear instead of quadratic. We thus use $2 * (2 * dim + 1)$ extra linear space. The pseudo-code for sparse closure is shown in Algorithm 9. The index is computed at the start of every iteration by calling the function $compute\_index$ at line 10. The pseudo code for $compute\_index$ is shown in Algorithm 10. The first element of row index r and column index c contains the size s of the index i.e., the number of finite entries for *k*-th row(column), the next *s* entries are the indices. $m_{i,j}$ is computed in Algorithm 9 only when the location of both $m_{i,k}$ and $m_{k,j}$ are present in the column and the row index respectively.

Algorithm 9 Sparse Closure

```
1.  function SPARSE_CLOSURE($m, r, c, t, dim$)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $r \leftarrow$ locations of finite values in k-th row
5.          $c \leftarrow$ locations of finite values in k-th column
6.          $t \leftarrow$ array to store diagonal elements
7.          $dim \leftarrow$ number of variables in program
8.      $n \leftarrow 2 * dim$
9.      for $k \leftarrow 0$ to $n$ do
10.         $compute\_index(result, k, r, c, dim)$
11.         for $i \leftarrow 0$ to $c_0$ do
12.             $i_1 \leftarrow c_{i+1}, ik \leftarrow m_{i_1,k}$
13.             for $j \leftarrow 0$ to $r_0$ do
14.                 $j_1 \leftarrow r_{j+1}, kj \leftarrow m_{k,j_1}$
15.                 $m_{i_1,j_1} \leftarrow min(m_{i_1,j_1}, ik + kj)$
16.     return $sparse\_strengthening(m, r, t, dim)$
```

Algorithm 10 Compute Index

```
1.  function COMPUTE_INDEX($m, k, r, c, dim$)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $k \leftarrow$ iteration number
5.          $r \leftarrow$ locations of finite values in k-th row
6.          $c \leftarrow$ locations of finite values in k-th column
7.          $dim \leftarrow$ number of variables in program
8.      $n \leftarrow 2 * dim, s \leftarrow 0$
9.      for $i \leftarrow 0$ to $n$ do
10.         if $is\_finite(m_{i,k})$ then
11.             $c_{s+1} \leftarrow i$
12.             $s \leftarrow s + 1$
13.     $c_0 \leftarrow s$
14.     $s \leftarrow 0$
15.     for $j \leftarrow 0$ to $n$ do
16.         if $is\_finite(m_{k,j})$ then
17.             $r_{s+1} \leftarrow j$
18.             $s \leftarrow s + 1$
19.     $r_0 \leftarrow s$
```

We also present a sparse algorithm for the strengthening step. Algorithm 11 shows the pseudo code of sparse strengthening. We store the indices of only the finite diagonal entries. The diagonal entries are again stored in an array to reduce TLB misses. Inside the double loop at line 14, $m_{i,j}$ is updated only when both $m_{i,\bar{i}}$ and $m_{\bar{j},j}$ are present in the diagonal index.

Algorithm 11 Sparse Strengthening

```
 1. function SPARSE_STRENGTHENING($m, d, t, dim$)
 2.     Parameters:
 3.         $m \leftarrow$ input matrix
 4.         $d \leftarrow$ index for diagonal elements
 5.         $t \leftarrow$ array to store diagonal elements
 6.         $dim \leftarrow$ number of variables in program
 7.     $n \leftarrow 2 * dim, s \leftarrow 0$
 8.     for $i \leftarrow 0$ to $n$ do
 9.         $t_i \leftarrow m_{i\oplus 1,i}$
10.         if $is\_finite(t_i)$ then
11.             $d_{s+1} \leftarrow i$
12.             $s \leftarrow s + 1$
13.     $d_0 \leftarrow s$
14.     for $i \leftarrow 0$ to $d_0$ do
15.         $i_1 \leftarrow d_{i+1}, ii \leftarrow t_{i_1}$
16.         for $j \leftarrow 0$ to $d_0$ do
17.             $j_1 \leftarrow d_{j+1}, jj \leftarrow t_{j_1}$
18.             $m_{i_1\oplus 1,j_1} \leftarrow min(m_{i_1\oplus 1,j_1}, (ii + jj)/2)$
19.     for $i \leftarrow 0$ to $n$ do
20.         if $m_{i,i} < 0$ then
21.             return $Bottom$
22.     return $m$
```

3.3. Half Closure

APRON [19] is a popular numerical static analysis library supporting the Octagon domain. It uses Floyd-Warshall to compute the triple loop of the Octagon closure. However, it stores only the lower triangular part of the matrix to save space as the remaining part can be recovered by coherence. The half representation is shown in Figure 3.2. An element with index $\{i, j\}$ is the $j + ((i + 1) * (i + 1))/2$-th element in the matrix . For all the half algorithms presented from now on, its assumed that $m_{i,j}$ returns the element $m_{j+((i+1)*(i+1))/2}$. The function $get\_element$ in Algorithm 12 returns an element from half matrix given the index. The pseudo code of APRON's Octagon closure algorithm is shown in Algorithm 13.

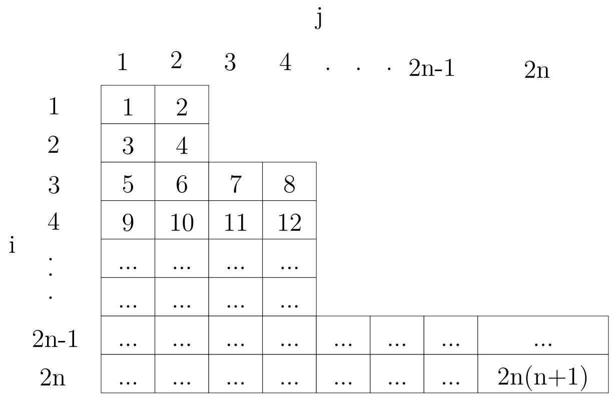

Figure 3.2.: Half representation of octagon matrix

Algorithm 12 Get Element in Half Representation

```
function GET_ELEMENT($m, i, j$)
    if $i < j$ then
        return $m_{j\oplus1,i\oplus1}$
    else
        return $m_{i,j}$
```

Although APRON only stores the lower triangular part of the matrix, it still performs the same number of operations for the triple loop as Floyd-Warshall on the full matrix. We now present an algorithm for the Octagon closure which also works with a half-representation but performs half the operations for the triple loop.

Algorithm 13 Closure APRON

```
function APRON_CLOSURE(m,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $k \leftarrow 0$ to $n$ do
        for $i \leftarrow 0$ to $n$ do
            $i_2 \leftarrow i \vee 1$
            $ik \leftarrow get\_element(m, i, k)$
            $ikk \leftarrow get\_element(m, i, k \oplus 1)$
            for $j \leftarrow 0$ to $i2$ do
                $kj \leftarrow get\_element(m, k, j)$
                $kkj \leftarrow get\_element(m, k \oplus 1, j)$
                $op \leftarrow min(ik + kj, ikk + kkj)$
                $m_{i,j} \leftarrow min(m_{i,j}, op)$
    return $strengthening\_apron(m, dim)$
```

Algorithm 14 Strengthening APRON

```
function STRENGTHENING_APRON(m,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $i \leftarrow 0$ to $n$ do
        $i_2 \leftarrow (i \vee 1) + 1$
        $ii \leftarrow (m_{i,i\oplus1})/2$
        for $j \leftarrow 0$ to $i_2$ do
            $jj \leftarrow (m_{j\oplus1,j})/2$
            $m_{i,j} \leftarrow min(m_{i,j}, ii + jj)$
    for $i \leftarrow 0$ to $n$ do
        if $m_{i,i} < 0$ then
            return $Bottom$
    return $m$
```

Algorithm 15 Compute Column Half Scalar

```
function COMPUTE_COL_HALF_SCALAR(m,c,d,t,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $c \leftarrow$ column to be modified
        $d \leftarrow$ column used for modifying
        $t \leftarrow$ array to store updated column values
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    if c is odd then
        $s \leftarrow c + 1$
    else
        $s \leftarrow c + 2$
    $kj \leftarrow m_{d,c}$
    for $i \leftarrow s$ to $n$ do
        $ik \leftarrow m_{i,d}$
        $m_{i,c} \leftarrow min(m_{i,c}, ik + kj)$
        $t_{i \oplus 1} \leftarrow m_{i,c}$
```

Algorithm 16 Compute Row Half Scalar

```
function COMPUTE_ROW_HALF_SCALAR(m,r,s,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $r \leftarrow$ row to be modified
        $s \leftarrow$ row used for modifying
        $dim \leftarrow$ number of variables in program
    if r is odd then
        $e \leftarrow r - 1$
    else
        $e \leftarrow r$
    $ik \leftarrow m_{r,s}$
    for $j \leftarrow 0$ to $e$ do
        $kj \leftarrow m_{s,j}$
        $m_{r,j} \leftarrow min(m_{r,j}, ik + kj)$
```

The APRON closure as shown in Algorithm 13 performs two min operations at lines 14 and 15 per iteration of the outermost loop. During the $2k$-th and $(2k + 1)$-th iterations, for a given element $m_{i,j}$, the algorithm performs updates using the same pair of elements $m_{i,2k}$, $m_{2k,j}$, and $m_{i,2k+1}$, $m_{2k+1,j}$ twice. To reduce the opcount to half, it is desirable to use these pairs only once. We can accomplish this by first computing the updated values for the elements in the $2k$ and $(2k + 1)$-th row and column that APRON will compute after $(2k + 1)$-th iteration. The updated elements in these rows and columns can then be used to compute updated values after the $(2k + 1)$-th iteration for the remaining elements in the matrix.

Algorithm 17 Compute Iteration Half Scalar

```
function COMPUTE_ITERATION_HALF_SCALAR(m,k,a,b,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $k \leftarrow$ iteration number
        $a \leftarrow$ column values for 2k-th column
        $b \leftarrow$ column values for (2k+1)-th column
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $i \leftarrow 0$ to $2k$ do
        $i_2 \leftarrow (i \vee 1) + 1$
        $br \leftarrow i_2 < 2k \ ? \ i_2 \ : \ 2k$
        $ik \leftarrow get\_element(m, i, 2k), ikk \leftarrow get\_element(m, i, 2k + 1)$
        for $j \leftarrow 0$ to $br$ do
            $kj \leftarrow m_{2k,j}, kkj \leftarrow m_{2k+1,j}$
            $m_{i,j} \leftarrow min(m_{i,j}, min(ik + kj, ikk + kkj))$
        for $j \leftarrow 2k + 2$ to $i_2$ do
            $kj \leftarrow a_j, kkj \leftarrow b_j$
            $m_{i,j} \leftarrow min(m_{i,j}, min(ik + kj, ikk + kkj))$
    for $i \leftarrow 2k + 2$ to $n$ do
        $i_2 \leftarrow (i \vee 1) + 1$
        $br \leftarrow i_2 < 2k \ ? \ i_2 \ : \ 2k$
        $ik \leftarrow m_{i,2k}, ikk \leftarrow m_{i,2k+1}$
        for $j \leftarrow 0$ to $br$ do
            $kj \leftarrow m_{2k,j}, kkj \leftarrow m_{2k+1,j}$
            $m_{i,j} \leftarrow min(m_{i,j}, min(ik + kj, ikk + kkj))$
        for $j \leftarrow 2k + 2$ to $i_2$ do
            $kj \leftarrow a_j, kkj \leftarrow b_j$
            $m_{i,j} \leftarrow min(m_{i,j}, min(ik + kj, ikk + kkj))$
```

Algorithm 18 Closure Half Scalar

```
1.  function CLOSURE_HALF_SCALAR(m,a,b,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $a \leftarrow$ array to store updated 2k-th column values
5.          $b \leftarrow$ array to store updated (2k+1)-th column values
6.          $dim \leftarrow$ number of variables in program
7.      for $k \leftarrow 0$ to $dim$ do
8.          $compute\_col\_half\_scalar(m, 2k, 2k+1, a, dim)$
9.          $compute\_col\_half\_scalar(m, 2k+1, 2k, b, dim)$
10.         $compute\_row\_half\_scalar(m, 2k, 2k+1, dim)$
11.         $compute\_row\_half\_scalar(m, 2k+1, 2k, dim)$
12.         $compute\_iteration\_half\_scalar(m, k, a, b, dim)$
13.     return $strengthening\_half\_scalar(m, a, dim)$
```

Algorithm 19 Strengthening Half Scalar

```
1.  function STRENGTHENING_HALF_SCALAR(m,t,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $t \leftarrow$ array to store diagonal elements
5.          $dim \leftarrow$ number of variables in program
6.      $n \leftarrow 2 * dim$
7.      for $i \leftarrow 0$ to $n$ do
8.          $t_i \leftarrow m_{i\oplus 1,i}$
9.      for $i \leftarrow 0$ to $n$ do
10.         $i_2 \leftarrow (i \vee 1) + 1$
11.         $ii \leftarrow t_{i\oplus 1}$
12.         for $j \leftarrow 0$ to $i_2$ do
13.             $jj \leftarrow t_j$
14.             $m_{i,j} \leftarrow min(m_{i,j}, (ii + jj)/2)$
15.     for $i \leftarrow 0$ to $n$ do
16.         if $m_{i,i} < 0$ then
17.             return $Bottom$
18.     return $m$
```

The pseudo code for the half closure is shown in Algorithm 18. During the k-th iteration, we first update $2k$ and $(2k+1)$-th row and column using Algorithms 15 and 16 respectively. The code at line 9 in Algorithm 15 sets s to $2k+2$. Similarly, the code at line 7 in Algorithm 16 sets e to $2k$. We skip computing elements $m_{2k,2k}, m_{2k,2k+1}, m_{2k+1,2k}$ and $m_{2k+1,2k+1}$ which is correct and is explained later. For Floyd-Warshall, the elements in k-th row and column do not change during k-th iteration. Thus, we update $2k$-th row (column) using $(2k+1)$-th row(column) and vice-versa.

The updated values are then used to compute the remaining elements using Algorithm 17. The coherence property insures that we can get missing values corresponding to the upper triangular part from the lower triangular part. At lines 17 and 27 in Algorithm 17, for $j \geq 2k+2$, the elements are in the upper triangular part and are accessed row-wise for an algorithm working with the full matrix. The corresponding elements in the lower triangular part for the half matrix are accessed column-wise due to coherence. To improve memory performance, we put the updated column values in an array as shown in Algorithm 15 at line 17 and access the updated values from the array during $compute\_iteration\_half\_scalar$ at lines 17 and 27.

The strengthening phase is the same as before except that we work with only the lower triangular part of the matrix. The strengthening phase is optimized to reduce TLB misses by storing diagonal elements in an array. The pseudo code for strengthening the half closure is shown in Algorithm 19.

Algorithm 20 Compute Element in Half Representation with AVX

```
1.  function COMPUTE_ELEM_HALF_AVX($ij, ik, ikk, kj, kkj$)
2.      $op_1 \leftarrow avx\_add\_double(ik, kj)$
3.      $op_2 \leftarrow avx\_add\_double(ikk, kkj)$
4.      $op_3 \leftarrow avx\_min\_double(op_1, op_2)$
5.      $op_4 \leftarrow avx\_min\_double(ij, op_3)$
6.      return $op_4$
```

Algorithm 21 Closure in Half Representation with AVX

```
function HALF_CLOSURE_AVX(m,a,b,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $a \leftarrow$ array to store updated 2k-th column
        $b \leftarrow$ array to store updated (2k+1)-th column
        $dim \leftarrow$ number of variables in program
    for $k \leftarrow 0$ to $dim$ do
        $compute\_col\_half\_scalar(m, 2k, 2k+1, a, dim)$
        $compute\_col\_half\_scalar(m, 2k+1, 2k, b, dim)$
        $compute\_row\_half\_scalar(m, 2k, 2k+1, dim)$
        $compute\_row\_half\_scalar(m, 2k+1, 2k, dim)$
        $compute\_iteration\_half\_avx(m, k, a, b, dim)$
    return $strengthening\_half\_avx(m, a, dim)$
```

Algorithm 22 Compute Iteration in Half Representation with AVX

```
function COMPUTE_ITERATION_HALF_AVX(m,k,a,b,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $k \leftarrow$ iteration number
        $a \leftarrow$ column values for 2k-th column values
        $b \leftarrow$ column values for (2k+1)-th column values
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $i \leftarrow 0$ to $2k$ do
        $l \leftarrow pad(2k+2)$
        $i_2 \leftarrow (i \vee 1) + 1$
        $br \leftarrow i_2 < 2k\ ?\ i_2\ :\ 2k$
        $ik \leftarrow avx\_set\_double(get\_element(m, i, 2k))$
        $ikk \leftarrow avx\_set\_double(get\_element(m, i, 2k+1))$
        for $j \leftarrow 0$ to $br/4$ do
            $ij \leftarrow avx\_load\_double(m_{i,j*4})$
            $kj \leftarrow avx\_load\_double(m_{2k,j*4})$
            $kkj \leftarrow avx\_load\_double(m_{2k+1,j*4})$
            $op \leftarrow compute\_elem\_half\_avx(ij, ik, ikk, kj, kkj)$
            $avx\_store\_double(m_{i,j*4}, op)$
        for $j \leftarrow l$ to $i_2/4$ do
            $ij \leftarrow avx\_load\_double(m_{i,j*4})$
            $kj \leftarrow avx\_load\_double(a_{j*4})$
            $kkj \leftarrow avx\_load\_double(b_{j*4})$
            $op \leftarrow compute\_elem\_half\_avx(ij, ik, ikk, kj, kkj)$
            $avx\_store\_double(m_{i,j*4}, op)$
    for $i \leftarrow 2k+2$ to $n$ do
        $i_2 \leftarrow (i \vee 1) + 1$
        $br \leftarrow i_2 < 2k\ ?\ i_2\ :\ 2k$
        $ik \leftarrow avx\_set\_double(m_{i,2k})$
        $ikk \leftarrow avx\_set\_double(m_{i,2k+1})$
        for $j \leftarrow 0$ to $br/4$ do
            $ij \leftarrow avx\_load\_double(m_{i,j*4})$
            $kj \leftarrow avx\_load\_double(m_{2k,j*4})$
            $kkj \leftarrow avx\_load\_double(m_{2k+1,j*4})$
            $op \leftarrow compute\_elem\_half\_avx(ij, ik, ikk, kj, kkj)$
            $avx\_store\_double(m_{i,j*4}, op)$
        for $j \leftarrow l$ to $i_2/4$ do
            $ij \leftarrow avx\_load\_double(m_{i,j*4})$
            $kj \leftarrow avx\_load\_double(a_{j*4})$
            $kkj \leftarrow avx\_load\_double(b_{j*4})$
            $op \leftarrow compute\_elem\_half\_avx(ij, ik, ikk, kj, kkj)$
            $avx\_store\_double(m_{i,j*4}, op)$
```

The half closure algorithm can also be vectorized using Intel's AVX intrinsics. The pseudo code for half closure with AVX is shown in Algorithm 21. It is not possible to vectorize the computation of $2k$ and $(2k+1)$-th column as elements are accessed column-wise. The computation of rows can be vectorized but does not yield much speedup as the computation is small. Major speedup is achieved by vectorizing computation of remaining elements using updated operands. The pseudo code for this computation using AVX is shown in Algorithm 22.

If br is not a multiple of 4 then after the end of loop at lines 15 and 32, we perform scalar computation of $m_{i,j}$ while $j \leq br$. The function pad at line 6 returns the nearest integer $l \geq 2k+2$ divisible by 4. If $2k+2$ is not divisible by 4 then again we perform scalar computation of $m_{i,j}$ from $j = 2k+2$ to $j = l-1$ before entering the loops at lines 21 and 38. After that we resume vector computation from $j \leftarrow l$. Similarly, if ii is not divisible by 4, we perform scalar computation of $m_{i,j}$ after the loops at lines 21 and 38. For simplicity, the scalar computations are not shown in Algorithm 22. Notice that storing the updated column values in an array enables vectorization of the loop at line 21 and 38 which would not have been possible with columns of the half matrix.

The strengthening can also be vectorized for half representation. The pseudo code for it is shown in Algorithm 23. Again, storing the diagonal values in arrays enables the vectorization of the double loop which otherwise would not have been possible.

Algorithm 23 Strengthening in Half Representation with AVX

```
function STRENGTHENING_HALF_AVX(m,t,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $t \leftarrow$ array to store diagonal elements
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $i \leftarrow 0$ to $n$ do
        $t_i \leftarrow m_{i\oplus 1,i}$
    for $i \leftarrow 0$ to $n$ do
        $i_2 \leftarrow (i \vee 1) + 1$
        $ii \leftarrow avx\_set\_double(t_{i\oplus 1})$
        for $j \leftarrow 0$ to $i_2/4$ do
            $jj \leftarrow avx\_load\_double(t_{j*4})$
            $op_1 \leftarrow avx\_load\_double(m_{i,j*4})$
            $op_2 \leftarrow avx\_add\_double(ii, jj)$
            $op_2 \leftarrow avx\_mul\_double(op_2, 0.5)$
            $op_3 \leftarrow avx\_min\_double(op_1, op_2)$
            $avx\_store\_double(m_{i,j*4}, op_3)$
    for $i \leftarrow 0$ to $n$ do
        if $m_{i,i} < 0$ then
            return $Bottom$
    return $m$
```

Theorem 3.3.1. *Let h and m be the half matrix obtained after applying Half closure and APRON closure respectively. Then, we have the $\forall i, j$, $m_{i,j} = h_{i,j}$.*

Proof. We show by induction on k that after applying half closure, h contains the same values for all elements as m after applying APRON closure. The argument holds for $k = 0$. Suppose the argument holds for $k = r-1$ i.e.,

$$\forall i, j, h_{i,j}^{r-1} = m_{i,j}^{(2r-1)} \tag{3.1}$$

(Notice half closure runs twice as fast as APRON closure). We now prove that the argument holds for $k = r$. Let us first assume absence of negative cycle.

First consider elements $h^r_{2r,2r+1}$ and $h^r_{2r+1,2r}$. For the r-th iteration both elements do not change for half closure. For APRON closure we have for $2r$-th iteration,

$$\begin{aligned} m^{2r}_{2r,2r+1} = min(&m^{2r-1}_{2r,2r+1}, \\ &m^{2r}_{2r,2r} + m^{2r-1}_{2r,2r+1}, \\ &m^{2r-1}_{2r,2r+1} + m^{2r-1}_{2r+1,2r+1}) \end{aligned} \tag{3.2}$$

Since there is no negative cycle at $2r$ and $2r+1$, we have,

$$m^{2r}_{2r,2r} = m^{2r-1}_{2r+1,2r+1} = 0 \tag{3.3}$$

Thus $m_{2r,2r+1}$ does not change during $2r$-th iteration. Now for $(2r+1)$-th iteration we have,

$$\begin{aligned} m^{2r+1}_{2r,2r+1} = min(&m^{2r-1}_{2r,2r+1}, \\ &m^{2r}_{2r,2r+1} + m^{2r-1}_{2r+1,2r+1}, \\ &m^{2r+1}_{2r,2r} + m^{2r-1}_{2r,2r+1}) \end{aligned} \tag{3.4}$$

Again we know that in absence of negative cycle, the elements $m_{2r,2r}$ and $m_{2r+1,2r+1}$ remain zero.

Hence $m^{2r+1}_{2r,2r+1} = m^{2r-1}_{2r,2r+1}$, it can be similarly shown that $m^{2r+1}_{2r+1,2r} = m^{2r-1}_{2r+1,2r}$. Now consider the elements in $2r$-th column. Half closure computes elements in $2r$-th column during r-th iteration as,

$$\begin{aligned} h^r_{i,2r} = min(&h^r_{i,2r}, \\ &h^{r-1}_{i,2r+1} + h^{r-1}_{2r+1,2r}) \end{aligned} \tag{3.5}$$

For APRON closure we have for $2r$-th iteration,

$$\begin{aligned} m^{2r}_{i,2r} = min(&m^{2r-1}_{i,2r}, \\ &m^{2r-1}_{i,2r} + m^{2r-1}_{2r,2r}, \\ &m^{2r-1}_{i,2r+1} + m^{2r-1}_{2r+1,2r}) \end{aligned} \tag{3.6}$$

On simplifying,

$$\begin{aligned} m^{2r}_{i,2r} = min(&m^{2r-1}_{i,2r}, \\ &m^{2r-1}_{i,2r+1} + m^{2r-1}_{2r+1,2r}) \end{aligned} \tag{3.7}$$

Similarly for elements in $(2r+1)$-th column during $2r$-th iteration we have,

$$\begin{aligned} m^{2r}_{i,2r+1} = min(&m^{2r-1}_{i,2r+1}, \\ &m^{2r}_{i,2r} + m^{2r-1}_{2r,2r+1}) \end{aligned} \tag{3.8}$$

Now for elements in column $2r$ we have during $(2r+1)$-th iteration,

$$\begin{aligned} m_{i,2r}^{2r+1} = min(&m_{i,2r}^{2r}, \\ &m_{i,2r+1}^{2r} + m_{2r+1,2r}^{2r-1}) \end{aligned} \tag{3.9}$$

Combining equation 3.7 and 3.8 we have,

$$\begin{aligned} m_{i,2r}^{2r+1} = min(&m_{i,2r}^{2r-1}, \\ &m_{i,2r+1}^{2r-1} + m_{2r+1,2r}^{2r-1}, \\ &m_{i,2r}^{2r} + m_{2r,2r+1}^{2r-1} + m_{2r+1,2r}^{2r-1}) \end{aligned} \tag{3.10}$$

Since there is no negative cycle at $2r$ we have,

$$m_{i,2r}^{2r} + m_{2r,2r+1}^{2r-1} + m_{2r+1,2r}^{2r-1} \geq m_{i,2r}^{2r} \tag{3.11}$$

.

Thus equation 3.10 becomes,

$$\begin{aligned} m_{i,2r}^{2r+1} = min(&m_{i,2r}^{2r}, \\ &m_{i,2r+1}^{2r-1} + m_{2r+1,2r}^{2r-1}) \end{aligned} \tag{3.12}$$

this expression is equivalent to expression for half closure in equation 3.5 due to equation 3.1, thus, $m_{i,2r}^{2r+1} = h_{i,2r}^{2r+1}$. Similarly it can be shown that elements in column $(2r+1)$ as well as in row $2r$ and $(2r+1)$ contain the same value after$(2r+1)$-th iteration as elements in half closure after r-th iteration. Thus, we have,

$$\begin{aligned} h_{*,2r}^{r} &= m_{(*,2r)}^{2r+1} \\ h_{*,2r+1}^{r} &= m_{(*,2r+1)}^{2r+1} \\ h_{2r,*}^{r} &= m_{(2r,*)}^{2r+1} \\ h_{2r+1,*}^{r} &= m_{(2r+1,*)}^{2r+1} \end{aligned} \tag{3.13}$$

Half closure updates remaining elements $h_{i,j}$ using updated row and column values which are the same as APRON closure after $(2r+1)$-th iteration. Thus, if for any element $h_{i,j}$ the shortest path goes through $2r$ or $2r+1$ it gets updated to the correct value. This shows that in absence of negative cycle,

$$\forall i, j, h_{i,j}^{r} = m_{i,j}^{(2r+1)} \tag{3.14}$$

In presence of negative cycle, APRON closure returns Bottom element by detecting if $m_{i,i} < 0$. We also update all the diagonal elements except $m_{2r,2r}$ and $m_{2r+1,2r+1}$ during r-th iteration. Thus, any negative cycle for $2r$ that does not pass through $(2r+1)$ gets detected and Bottom element is returned. If there is negative cycles for $2r$ that goes through $(2r+1)$ it gets detected during strengthening step as we have for strengthening,

$$h_{2r,2r} = min(h_{2r,2r}, (h_{2r,2r+1} + h_{2r+1,2r})/2) \tag{3.15}$$

.

Thus, $\forall i, j$, $h^r_{i,j} = m^{(2r+1)}_{i,j}$ in presence of negative cycle also.

□

3.4. Half Sparse Closure

Algorithm 24 Half Sparse Closure

```
1.  function CLOSURE_HALF_SPARSE(m,r,r',c,c',a,b, dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $r \leftarrow$ locations of finite values in 2k-th row
5.          $r' \leftarrow$ locations of finite values in (2k+1)-th row
6.          $c \leftarrow$ locations of finite values in 2k-th column
7.          $c' \leftarrow$ locations of finite values in (2k+1)-th column
8.          $a \leftarrow$ array to store updated 2k-th column values
9.          $b \leftarrow$ array to store updated (2k+1)-th column values
10.         $dim \leftarrow$ number of variables in program
11.     for $k \leftarrow 0$ to $dim$ do
12.         $compute\_index\_half\_sparse(m, k, r, r', c, c', dim)$
13.         $compute\_col\_half\_sparse(m, 2k, 2k+1, c, c', a, dim)$
14.         $compute\_col\_half\_sparse(m, 2k+1, 2k, c', c, b, dim)$
15.         $compute\_row\_half\_sparse(m, 2k, 2k+1, r, r', dim)$
16.         $compute\_row\_half\_sparse(m, 2k+1, 2k, r', r, dim)$
17.         $compute\_iteration\_half\_sparse(m, k, r, r', c, c', a, b, dim)$
18.     return $strengthening\_half\_sparse(m, r, a, dim)$
```

Algorithm 25 Compute Index Half Sparse

```
1.  function COMPUTE_INDEX_HALF_SPARSE(m,k,r,r',c,c',dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $k \leftarrow$ iteration number
5.          $r \leftarrow$ locations of finite values in 2k-th row
6.          $r' \leftarrow$ locations of finite values in (2k+1)-th row
7.          $c \leftarrow$ locations of finite values in 2k-th column
8.          $c' \leftarrow$ locations of finite values in (2k+1)-th column
9.          $dim \leftarrow$ number of variables in program
10.     $n \leftarrow 2 * dim, s_1 \leftarrow 0, s_2 \leftarrow 0$
11.     for $i \leftarrow 2k+2$ to $n$ do
12.         if $is\_finite(m_{i,2k})$ then
13.             $c_{s_1+1} \leftarrow i, s_1 \leftarrow s_1 + 1$
14.         if $is\_finite(m_{i,2k+1})$ then
15.             $c'_{s_2+1} \leftarrow i, s_2 \leftarrow s_2 + 1$
16.     $c_0 \leftarrow s_1, c'_0 \leftarrow s_2$
17.     $s_1 \leftarrow 0, s_2 \leftarrow 0$
18.     for $j \leftarrow 0$ to $2k$ do
19.         if $is\_finite(m_{2k,j})$ then
20.             $r_{s_1+1} \leftarrow j, s_1 \leftarrow s_1 + 1$
21.         if $is\_finite(m_{2k+1,j})$ then
22.             $r'_{s_2+1} \leftarrow j, s_2 \leftarrow s_2 + 1$
23.     $r_0 \leftarrow s_1, r'_0 \leftarrow s_2$
```

Half Sparse Closure combines the benefits of a reduced opcount as in half closure and the benefits of the sparse structure of a matrix as in sparse closure. We use the same sparse index arrays for storing locations of finite values in half sparse closure as for sparse closure. Algorithm 24 shows the pseudo code for half sparse closure.

At the start of the k-th iteration of the outermost loop, we compute the locations of finite values for $2k$ and $(2k+1)$-th row and column using the function $compute\_index\_half\_sparse$ shown in Algorithm 25. The indices are then used to update the elements in $2k$ and $(2k+1)$-th row and column using functions $compute\_row\_half\_sparse$ and $compute\_col\_half\_sparse$ shown in Algorithms 27 and 26 respectively. If during the update, a non finite constraint becomes finite, its location is added to the corresponding index. The indices obtained after $2k$ and $(2k+1)$-th row and column are used for the computation of the remaining elements using the function $compute\_iteration\_half\_sparse$ shown in Algorithm 28.

Algorithm 26 Compute Column Half Sparse

```
1.  function COMPUTE_COL_HALF_SPARSE(m,k,kk,c,c',t,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $k \leftarrow$ column to be modified
5.          $kk \leftarrow$ column used for modifying
6.          $c \leftarrow$ index of column to be modified
7.          $c' \leftarrow$ index of column used for modifying
8.          $t \leftarrow$ array to store updated column values
9.          $dim \leftarrow$ number of variables in program
10.     $n \leftarrow 2 * dim$
11.     if $is\_finite(m_{kk,k})$ then
12.         $s_1 \leftarrow c_0, s_2 \leftarrow c'_0$
13.         for $i \leftarrow 0$ to $s_2$ do
14.             $i_1 \leftarrow c'_{i+1}$
15.             if $is\_finite(m_{i_1,k})$ then
16.                 $m_{i_1,k} \leftarrow min(m_{i_1,k}, m_{i_1,kk} + m_{kk,k})$
17.             else
18.                 $m_{i_1,k} \leftarrow m_{i_1,kk} + m_{kk,k}$
19.                 $c_{s_1+1} \leftarrow i_1, s_1 \leftarrow s_1 + 1$
20.         $c_0 \leftarrow s_1$
21.     for $i \leftarrow 2k+2$ to $n$ do
22.         $t_i \leftarrow m_{i,k}$
```

Algorithm 27 Compute Row Half Sparse

```
1.  function COMPUTE_ROW_HALF_SPARSE(m,k,kk,r,r',dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $k \leftarrow$ row to be modified
5.          $kk \leftarrow$ row used for modifying
6.          $r \leftarrow$ index of row to be modified
7.          $r' \leftarrow$ index of row used for modifying
8.          $dim \leftarrow$ number of variables in program
9.      if $is\_finite(m_{k,kk})$ then
10.         $s_1 \leftarrow r_0, s_2 \leftarrow r'_0$
11.         for $j \leftarrow 0$ to $s_2$ do
12.             $j_1 \leftarrow r'_{j+1}$
13.             if $is\_finite(m_{k,j_1})$ then
14.                 $m_{k,j_1} \leftarrow min(m_{k,j_1}, m_{k,kk} + m_{kk,j_1})$
15.             else
16.                 $m_{k,j_1} \leftarrow m_{k,kk} + m_{kk,j_1}$
17.                 $r_{s_1+1} \leftarrow j_1, s_1 \leftarrow s_1 + 1$
18.         $r_0 \leftarrow s_1$
```

Notice that now we need four arrays for storing indices instead of two for sparse closure. The first element of each array stores the size, and maximum size of each array can be $2*dim$. Thus, we need $4*(2*dim+1)$ extra space which is linear with respect to the number of variables.

Algorithm 28 Compute Iteration Half Sparse

```
 1. function COMPUTE_ITERATION_HALF_SPARSE(m,k,r,r',c,c',a,b,dim)
 2.     Parameters:
 3.         $m \leftarrow$ input matrix
 4.         $k \leftarrow$ Iteration number of outer loop for Floyd-Warshall
 5.         $r \leftarrow$ locations of finite values in 2k-th row
 6.         $r' \leftarrow$ locations of finite values in (2k+1)-th row
 7.         $c \leftarrow$ locations of finite values in 2k-th column
 8.         $c' \leftarrow$ locations of finite values in (2k+1)-th column
 9.         $a \leftarrow$ column values for 2k-th column
10.         $b \leftarrow$ column values for (2k+1)-th column
11.         $dim \leftarrow$ number of variables in program
12.     for $i \leftarrow 0$ to $r_0$ do
13.         $i_1 \leftarrow r_{i+1}, ik \leftarrow m_{2k,i_1}$
14.         for $j \leftarrow 0$ to $r'_0$ do
15.             $j_1 \leftarrow r'_{j+1}, kj \leftarrow m_{2k+1,j_1}$
16.             $m_{i_1\oplus 1,j_1} \leftarrow min(m_{i_1\oplus 1,j_1}, ik+kj)$
17.         for $j \leftarrow 0$ to $c_0$ do
18.             $j_1 \leftarrow c_{j+1}, kj \leftarrow b_{j_1}$
19.             $m_{i_1\oplus 1,j_1\oplus 1} \leftarrow min(m_{i_1\oplus 1,j_1\oplus 1}, ik+kj)$
20.     for $i \leftarrow 0$ to $r'_0$ do
21.         $i_1 \leftarrow r'_{i+1}, ikk \leftarrow m_{2k+1,i_1}$
22.         for $j \leftarrow 0$ to $r_0$ do
23.             $j_1 \leftarrow r_{j+1}, kkj \leftarrow m_{2k,j_1}$
24.             $m_{i_1\oplus 1,j_1} \leftarrow min(m_{i_1\oplus 1,j_1}, ikk+kkj)$
25.         for $j \leftarrow 0$ to $c'_0$ do
26.             $j_1 \leftarrow c'_{j+1}, kkj \leftarrow a_{j_1}$
27.             $m_{i_1\oplus 1,j_1\oplus 1} \leftarrow min(m_{i_1\oplus 1,j_1\oplus 1}, ikk+kkj)$
28.     for $i \leftarrow 0$ to $c'_0$ do
29.         $i_1 \leftarrow c'_{i+1}, ik \leftarrow m_{i_1,2k+1}$
30.         for $j \leftarrow 0$ to $r'_0$ do
31.             $j_1 \leftarrow r'_{j+1}, kj \leftarrow m_{2k+1,j_1}$
32.             $m_{i_1,j_1} \leftarrow min(m_{i_1,j_1}, ik+kj)$
33.         for $j \leftarrow 0$ to $c_0$ do
34.             $j_1 \leftarrow c_{j+1}, kj \leftarrow b_{j_1}$
35.             $m_{i_1,j_1\oplus 1} \leftarrow min(m_{i_1,j_1\oplus 1}, ik+kj)$
36.     for $i \leftarrow 0$ to $c_0$ do
37.         $i_1 \leftarrow c_{i+1}, ikk \leftarrow m_{i_1,2k}$
38.         for $j \leftarrow 0$ to $r_0$ do
39.             $j_1 \leftarrow r_{j+1}, kkj \leftarrow m_{2k,j_1}$
40.             $m_{i_1,j_1} \leftarrow min(m_{i_1,j_1}, ikk+kkj)$
41.         for $j \leftarrow 0$ to $c'_0$ do
42.             $j_1 \leftarrow c_{j+1}, kkj \leftarrow a_{j_1}$
43.             $m_{i_1,j_1\oplus 1} \leftarrow min(m_{i_1,j_1\oplus 1}, ikk+kkj)$
```

The indices are updated at lines 19 and 17 in Algorithm 26 and 27 respectively. An alternative strategy could be to first update the columns (rows) without any index information and then compute the index. In this way, there is no need to update the index in case a non finite constraint becomes finite. However, computing the index first is advantageous because we only compute those row (column) elements that have finite operands which are usually low. As in the case of half closure, we keep the updated column values in an array to enhance memory performance.

Once the row and column elements are updated, the indices do not change. We compute an element $m_{i,j}$ only if both operands are present in the index. For the loops at lines 12 and 20 in Algorithm 28 we have $i < 2k$, thus operand ik and ikk correspond to the upper triangular part of a full matrix. By coherence, for half matrix, these elements can be accessed row-wise from the lower triangular part. Therefore, we use row indices r and r'. If we get i_1 from row index, it means row index of element to be computed should be $i_1 \oplus 1$ from coherence. Similarly, loops at lines 17, 25, 33 and 41 have $j \geq 2k+2$ hence these are accessed column-wise in half matrix. Therefore, we use column indices c and c'. If we get j_1 from the column index, it means the column index of the element to be computed should be $j_1 \oplus 1$ (by coherence).

For simplicity, some border cases are omitted in Algorithm 28. Specifically, we compute $i_2 \leftarrow (i_1 \vee 1) + 1$ and $br \leftarrow i_2 < 2k \; ? \; i_2 : \; 2k$ at the start of each i-loop. The loops at lines 14, 22, 30 and 38 in Algorithm 28 terminate if $j_1 > br$. Similarly, loops at lines 17, 25, 33 and 41 terminate if $j_1 > i_2$.

Algorithm 29 shows the sparse strengthening for the half matrix. The algorithm is similar to the sparse strengthening except that now we work with only the lower triangular part.

Algorithm 29 Strengthening Half Sparse

```
1.  function STRENGTHENING_HALF_SPARSE(m,d,t,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $d \leftarrow$ index for diagonal elements
5.          $t \leftarrow$ array to store diagonal elements
6.          $dim \leftarrow$ number of variables in program
7.      $n \leftarrow 2 * dim, s \leftarrow 0$
8.      for $i \leftarrow 0$ to $n$ do
9.          $t_i \leftarrow m_{i\oplus 1,i}$
10.         if $is\_finite(t_i)$ then
11.             $d_{s+1} \leftarrow i$
12.             $s \leftarrow s+1$
13.     $d_0 \leftarrow s$
14.     for $i \leftarrow 0$ to $d_0$ do
15.         $i_1 \leftarrow d_{i+1}$
16.         $ii \leftarrow t_{i_1}$
17.         for $j \leftarrow 0$ to $d_0$ do
18.             $j_1 \leftarrow d_{j+1}$
19.             $jj \leftarrow t_{j_1}$
20.             $m_{i_1\oplus 1,j_1} \leftarrow min(m_{i_1\oplus 1,j_1}, (ii+jj)/2)$
21.     for $i \leftarrow 0$ to $n$ do
22.         if $m_{i,i} < 0$ then
23.             return $Bottom$
24.     return $m$
```

3.5. Incremental Closure

Let *n* be the number of rows or columns in an Octagon matrix and $S = \{n_1, n_2, \ldots n_k\}$ be the set of non-negative integers such that $\forall i, n_i < n$ and $\forall i, j, \notin S$ we have, $m_{i,j} = m^{closed}_{i,j}$. A matrix satisfying the above condition is defined as "almost closed" as it is closed except for few rows and columns. If $V = \{V_{n_1}, V_{n_2}, \ldots V_{n_k}\}$ is the set of variables corresponding to S then it means that only the constraints involving at least one of the variables in *V* have changed. This is very useful in modeling assignment statements occurring in programs. Incremental closure can be used to compute Octagon closure of "almost closed" matrices quickly.

Algorithm 30 APRON Incremental Closure

```
function APRON_INCREMENTAL_CLOSURE(m,v,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $v \leftarrow$ index of modified variable
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $k \leftarrow 0$ to $n$ do
        for $i \leftarrow 2v$ to $2v+2$ do                    ▷ Row containing variable is updated
            $i_2 \leftarrow (i \vee 1) + 1$
            $ik \leftarrow get\_element(m,i,k), ikk \leftarrow get\_element(m,i,k \oplus 1)$
            for $j \leftarrow 0$ to $i2$ do
                $kj \leftarrow get\_element(m,k,j), kkj \leftarrow get\_element(m,k \oplus 1,j)$
                $m_{i,j} \leftarrow min(m_{i,j}, min(ik+kj, ikk+kkj))$
        for $j \leftarrow 2v$ to $2v+2$ do                    ▷ Column containing variable is updated
            $kj \leftarrow get\_element(m,k,j), kkj \leftarrow get\_element(m,k \oplus 1,j)$
            for $i \leftarrow j$ to $n$ do
                $ik \leftarrow get\_element(m,i,k), ikk \leftarrow get\_element(m,i,k \oplus 1)$
                $m_{i,j} \leftarrow min(m_{i,j}, min(ik+kj, ikk+kkj))$
    for $k \leftarrow 2v$ to $2v+2$ do                        ▷ Rest of the inequalities are updated
        for $i \leftarrow 0$ to $n$ do
            $i_2 \leftarrow (i \vee 1) + 1$
            $ik \leftarrow get\_element(m,i,k), ikk \leftarrow get\_element(m,i,k \oplus 1)$
            for $j \leftarrow 0$ to $i2$ do
                $kj \leftarrow get\_element(m,k,j), kkj \leftarrow get\_element(m,k \oplus 1,j)$
                $m_{i,j} \leftarrow min(m_{i,j}, min(ik+kj, ikk+kkj))$
    return $strengthening\_apron(m, dim)$
```

3.5.1. Dense Incremental Closure

APRON applies the incremental closure on the half representation of the Octagon matrix. Algorithm 30 shows the pseudo code for incremental closure in APRON for the case when constraints involving only one variable have changed. The complexity of incremental closure for one variable is quadratic.

In the algorithm, v is the index of the variable for which the constraints are not closed. The first n iterations of Floyd-Warshall are performed to update the elements in $2v$ and $(2v+1)$-th row and column. The updated values are used to update the rest of the elements in the matrix by performing $2v$ and $(2v+1)$-th iteration of Floyd-Warshall.

As with full closure, we found that the opcount of the incremental closure algorithm in APRON can be reduced in half. The first triple loop at line 7 in Algorithm 30 updates the elements in $2v$ and $(2v+1)$-th row and column. It can be seen that for a given element $m_{i,j}$, during $2k$ and $(2k+1)$-th iteration, the same elements $m_{i,2k}$, $m_{i,2k+1}$, $m_{2k,j}$, $m_{2k+1,j}$ are used for updates twice. Except for $k = 2v$ and $k = 2v+1$-th iteration the operands are not modified. Thus, value of operands for the min operation for $2k$ and $(2k+1)$-th iteration is the same when $k \neq v$. We know from the Half closure that for $2v$ and $(2v+1)$-th iteration, elements in $2v$ and $(2v+1)$-th row and column can be computed using half the operations. We thus reduce the number of iterations of the outermost loop at line 7 to half by having the outermost loop run twice as fast as APRON.

Algorithm 31 Dense Incremental Closure Scalar

```
1.  function INCREMENTAL_HALF_DENSE_SCALAR(m,v,a,b,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $v \leftarrow$ index of modified variable
5.          $a \leftarrow$ array to store updated 2v-th column values
6.          $b \leftarrow$ array to store updated (2v+1)-th column values
7.          $dim \leftarrow$ number of variables in program
8.          $n \leftarrow 2 * dim, br \leftarrow 2k < 2v?2k : 2v$
9.      for $k \leftarrow 0$ to $dim$ do
10.         for $i \leftarrow 2v$ to $2v + 2$ do
11.             $ik \leftarrow get\_element(m, i, 2k)$
12.             $ikk \leftarrow get\_element(m, i, 2k + 1)$
13.             for $j \leftarrow 0$ to $br$ do
14.                 $kj \leftarrow m_{2k,j}$
15.                 $kkj \leftarrow m_{2k+1,j}$
16.                 $m_{i,j} \leftarrow min(m_{i,j}, min(ik + kj, ikk + kkj))$
17.             for $j \leftarrow br$ to $2v$ do
18.                 $kj \leftarrow get\_element(m, 2k, j)$
19.                 $kkj \leftarrow get\_element(m, 2k + 1, j)$
20.                 $m_{i,j} \leftarrow min(m_{i,j}, min(ik + kj, ikk + kkj))$
21.         for $j \leftarrow 2v$ to $2v + 2$ do
22.             $kj \leftarrow get\_element(m, 2k, j)$
23.             $kkj \leftarrow get\_element(m, 2k + 1, j)$
24.             for $i \leftarrow 2v$ to $n$ do
25.                 $ik \leftarrow get\_element(m, i, 2k)$
26.                 $ikk \leftarrow get\_element(m, i, 2k + 1)$
27.                 $m_{i,j} \leftarrow min(m_{i,j}, min(ik + kj, ikk + kkj))$
28.     $compute\_col\_half\_scalar(m, 2v, 2v + 1, a, dim)$
29.     $compute\_col\_half\_scalar(m, 2v + 1, 2v, b, dim)$
30.     $compute\_row\_half\_scalar(m, 2v, 2v + 1, dim)$
31.     $compute\_row\_half\_scalar(m, 2v + 1, 2v, dim)$
32.     $compute\_iteration\_half\_scalar(m, v, a, b, dim)$
33.     return $strengthening\_half\_scalar(m, a, dim)$
```

The second triple loop at line 19 modifies the entire matrix using updated values of elements in $2v$ and $(2v + 1)$-th row and column. This is equivalent to performing two iterations of Floyd-Warshall and its opcount can be reduced in half by first computing $2v$ and $(2v + 1)$-th row and column as in Half closure.

Algorithm 31 shows the pseudo code for the dense incremental closure. We compute the triple loop at line 9 using half the operations. The rest of the algorithm is similar to the v-th iteration of half closure and can be computed using $compute\_col\_half\_scalar$, $compute\_row\_half\_scalar$ and $compute\_iteration\_half\_scalar$.

The dense incremental closure can also be vectorized using AVX. The pseudo code is shown in Algorithm 32. The loop at line 13 is vectorized while loop at line 23 is not as some of the elements (kj and kkj) correspond to the upper triangular part and are accessed column-wise in half matrix. Storing columns containing operands in an array incurs overhead as only two columns are updated instead of the complete matrix. The columns need to be stored again for the next iteration. Therefore, our trick of storing column values in an array which enables vectorization does not work. For the loop at line 27, elements are not accessed in the order suitable for vectorization and hence the loop is not vectorized. The rest of the computation is same as one iteration of a vectorized half closure and is vectorized the same way.

Algorithm 32 Dense Incremental Closure with AVX

```
function INCREMENTAL_HALF_DENSE_AVX(m,v,a,b,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $v \leftarrow$ index of modified variable
        $a \leftarrow$ array to store updated 2v-th column values
        $b \leftarrow$ array to store updated (2v+1)-th column values
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim, br \leftarrow 2k < 2v?2k : 2v$
    for $k \leftarrow 0$ to $dim$ do
        for $i \leftarrow 2v$ to $2v + 2$ do
            $ik \leftarrow avx\_set\_double(get\_element(m, i, 2k))$
            $ikk \leftarrow avx\_set\_double(get\_element(m, i, 2k))$
            for $j \leftarrow 0$ to $br/4$ do
                $kj \leftarrow avx\_load\_double(m_{2k,j*4})$
                $kkj \leftarrow avx\_load\_double(m_{2k+1,j*4})$
                $ij \leftarrow avx\_load\_double(m_{i,j*4})$
                $op_1 \leftarrow avx\_add\_double(ik, kj)$
                $op_2 \leftarrow avx\_add\_double(ikk, kkj)$
                $op_3 \leftarrow avx\_min\_double(op_1, op_2)$
                $op_4 \leftarrow avx\_min\_double(ij, op_3)$
                $avx\_store\_double(m_{i,j*4}, op_4)$
            $ik \leftarrow get\_element(m, i, 2k), ikk \leftarrow get\_element(m, i, 2k + 1)$
            for $j \leftarrow br$ to $2v$ do
                $kj \leftarrow get\_element(m, 2k, j)$
                $kkj \leftarrow get\_element(m, 2k + 1, j)$
                $m_{i,j} \leftarrow min(m_{i,j}, min(ik + kj, ikk + kkj))$
        for $j \leftarrow 2v$ to $2v + 2$ do
            $kj \leftarrow get\_element(m, 2k, j), kkj \leftarrow get\_element(m, 2k + 1, j)$
            for $i \leftarrow 2v$ to $n$ do
                $ik \leftarrow get\_element(m, i, 2k)$
                $ikk \leftarrow get\_element(m, i, 2k + 1)$
                $m_{i,j} \leftarrow min(m_{i,j}, min(ik + kj, ikk + kkj))$
    $compute\_col\_half\_scalar(m, 2v, 2v + 1, a, dim)$
    $compute\_col\_half\_scalar(m, 2v + 1, 2v, b, dim)$
    $compute\_row\_half\_scalar(m, 2v, 2v + 1, dim)$
    $compute\_row\_half\_scalar(m, 2v + 1, 2v, dim)$
    $compute\_iteration\_half\_avx(m, v, a, b, dim)$
    return $strengthening\_half\_avx(m, a, dim)$
```

3.5.2. Sparse Incremental Closure

The sparse incremental closure takes advantage of the sparse structure of the input matrix for incremental closure. It uses the same sparse indices as for Half closure. Thus, the extra memory overhead is the same. Algorithm 33 shows the pseudo code for the sparse incremental closure.

There is no need to compute an index for first triple loop. The reason is the same for not storing column values in an array. As the operands change in every iteration of the outermost loop, we would have to compute an index for each k in order to have only linear space overhead. But computing indices for each k would incur quadratic time overhead. An alternative strategy could be to compute an index for the whole matrix but that would increase space overhead to quadratic. The index is constructed only for the second triple loop. The computation in this loop is equivalent to a single iteration of Half Sparse Closure.

Algorithm 33 Sparse Incremental Closure

```
function INCREMENTAL_HALF_SPARSE(m,v,r,r',c,c',a,b,dim)
    Parameters:
        $m \leftarrow$ input matrix
        $v \leftarrow$ index of modified variable
        $r \leftarrow$ locations of finite values in 2k-th row
        $r' \leftarrow$ locations of finite values in (2k+1)-th row
        $c \leftarrow$ locations of finite values in 2k-th column
        $c' \leftarrow$ locations of finite values in (2k+1)-th column
        $a \leftarrow$ array to store updated 2v-th column values
        $b \leftarrow$ array to store updated (2v+1)-th column values
        $dim \leftarrow$ number of variables in program
    $n \leftarrow 2 * dim$
    for $k \leftarrow 0$ to $dim$ do
        for $i \leftarrow 2v$ to $2v+2$ do
            if $is\_finite(get\_element(m, i, 2k))$ then
                $ik \leftarrow get\_element(m, i, 2k)$
                for $j \leftarrow 0$ to $2v$ do
                    $kj \leftarrow get\_element(m, 2k, j)$
                    $m_{i,j} \leftarrow min(m_{i,j}, ik + kj)$
            if $is\_finite(get\_element(m, i, 2k+1))$ then
                $ikk \leftarrow get\_element(m, i, 2k+1)$
                for $j \leftarrow 0$ to $2v$ do
                    $kkj \leftarrow get\_element(m, 2k+1, j)$
                    $m_{i,j} \leftarrow min(m_{i,j}, ikk + kkj)$
        for $j \leftarrow 2v$ to $2v+2$ do
            if $is\_finite(get\_element(m, 2k, j))$ then
                $kj \leftarrow get\_element(m, 2k, j)$
                for $i \leftarrow 2v$ to $n$ do
                    $ik \leftarrow get\_element(m, i, 2k)$
                    $m_{i,j} \leftarrow min(m_{i,j}, ik + kj)$
            if $is\_finite(get\_element(m, 2k+1, j))$ then
                $kkj \leftarrow get\_element(m, 2k+1, j)$
                for $i \leftarrow j$ to $n$ do
                    $ikk \leftarrow get\_element(m, i, 2k+1)$
                    $m_{i,j} \leftarrow min(m_{i,j}, ikk + kkj)$
    $compute\_index\_half\_sparse(m, r, r', c, c', v, dim)$
    $compute\_col\_half\_sparse(m, 2v, 2v+1, c, c', a, dim)$
    $compute\_col\_half\_sparse(m, 2v+1, 2v, c', c, b, dim)$
    $compute\_row\_half\_sparse(m, 2v, 2v+1, r, r', dim)$
    $compute\_row\_half\_sparse(m, 2v+1, 2v, r', r, dim)$
    $compute\_iteration\_half\_sparse(m, v, r, r', c, c', a, b, dim)$
    return $strengthening\_half\_sparse(m, r, a, dim)$
```

4 Octagon Operators

Besides closure, the Octagon domain defines various other operators and transfer functions to handle various statements occurring commonly in programs such as assignment, loops, guard statements, etc. In this section, we present these operators and discuss how they can be made faster. It should be noted that unlike the closure, we do not implement sparse algorithms for these operators. Maintaining sparse index throughout the analysis will require us to store additional data structure which may take quadratic space in the worst case. As the runtime of the analysis is mainly dominated by the closure and the memory allocation, we do not maintain sparse index for other operators. In the rest of this chapter, we assume a half matrix and a function *oct_closure* to close the matrix (this can be any of the closure algorithms we described in the last chapter).

4.1. Join

The Join is a key operator for any abstract domain. Semantically, it involves combining information at locations in the control flow graph of the program where different branches merge. For octagons, the computation of join involves taking the element wise maximum of closed matrices. The matrix obtained by the join of two closed matrices is also closed. Algorithm 34 shows the pseudo code for the join operator.

Algorithm 34 Join Operator

```
1.  function JOIN(l,m,o, dim)
2.      Parameters:
3.          $l, m \leftarrow$ input matrix
4.          $o \leftarrow$ output matrix
5.          $dim \leftarrow$ number of variables in program
6.      $s \leftarrow 2 * dim * (dim + 1)$
7.      $oct\_closure(l, dim)$
8.      $oct\_closure(m, dim)$
9.      for $i \leftarrow 0$ to $s$ do
10.         $o_i \leftarrow max(l_i, m_i)$
```

Algorithm 35 Join Operator with AVX

```
1.  function JOIN_AVX(l,m,o, dim)
2.      Parameters:
3.          $l, m \leftarrow$ input matrix
4.          $o \leftarrow$ output matrix
5.          $dim \leftarrow$ number of variables in program
6.      $s \leftarrow 2 * dim * (dim + 1)$
7.      $oct\_closure(l, dim)$
8.      $oct\_closure(m, dim)$
9.      for $i \leftarrow 0$ to $s/4$ do
10.         $op_1 \leftarrow avx\_load\_double(l_{i*4})$
11.         $op_2 \leftarrow avx\_load\_double(m_{i*4})$
12.         $op_3 \leftarrow avx\_max\_double(op_1, op_2)$
13.         $avx\_store\_double(o_{i*4}, op_3)$
```

We vectorize the loop at line 9 in Algorithm 34 using AVX intrinsics. The vectorized join operator is shown in Algorithm 35. If s is not a multiple of 4, we have to compute the remaining elements using scalar computations. For simplicity, it has been omitted from Algorithm 34.

4.2. Meet

Meet is usually applied at guard statements of the program. It is used to keep the common information between the elements. There are three kind of meet operation defined for octagons. We describe meet with octagon next while the other two meet are defined later (see appendix A).

Algorithm 36 Meet Octagon

```
1.  function MEET_OCTAGONS(l,m,o, dim)
2.      Parameters:
3.          $l, m \leftarrow$ input matrix
4.          $o \leftarrow$ output matrix
5.          $dim \leftarrow$ number of variables in program
6.      $s \leftarrow 2 * dim * (dim + 1)$
7.      for $i \leftarrow 0$ to $s$ do
8.          $o_i \leftarrow min(l_i, m_i)$
```

Algorithm 37 Meet Octagon with AVX

```
1.  function MEET_OCTAGONS_AVX(l,m,o, dim)
2.      Parameters:
3.          $l, m \leftarrow$ input matrix
4.          $o \leftarrow$ output matrix
5.          $dim \leftarrow$ number of variables in program
6.      $s \leftarrow 2 * dim * (dim + 1)$
7.      for $i \leftarrow 0$ to $s/4$ do
8.          $op_1 \leftarrow avx\_load\_double(l_{i*4})$
9.          $op_2 \leftarrow avx\_load\_double(m_{i*4})$
10.         $op_3 \leftarrow avx\_min\_double(op_1, op_2)$
11.         $avx\_store\_double(o_{i*4}, op_3)$
```

The meet operation between two octagons involves taking the element wise minimum of matrices. It does not require the matrices to be closed. However, the matrix obtained by the meet of two octagon matrices is not necessarily closed. Algorithm 36 shows the pseudo code for the meet operation between two octagons.

We vectorize the loop at line 7 in Algorithm 36 using AVX intrinsics. The vectorized pseudo code is shown in Algorithm 37. As for the join, for presentation purposes, we omit the cases when s is not a multiple of 4.

4.3. Inclusion Testing

The inclusion operator is used to test if the set of concrete objects represented by an octagon is already contained inside a set of concrete objects represented by another octagon. It is used to form an ordering between different elements in the Octagonal lattice. The octagons with more information are higher in the lattice. It involves closing the first matrix and then an element wise comparison with the second matrix. If one of the element in the first matrix is greater than the corresponding element in the second matrix then the test fails. Algorithm 38 shows the pseudo code for the inclusion testing operator.

Algorithm 38 Inclusion Testing

```
 1. function IS_INCLUDED(l,m, dim)
 2.     Parameters:
 3.         $l, m \leftarrow$ input matrix
 4.         $dim \leftarrow$ number of variables in program
 5.     $s \leftarrow 2 * dim * (dim + 1)$
 6.     $n \leftarrow 2 * dim$
 7.     $oct\_closure(l, dim)$
 8.     for $i \leftarrow 0$ to $s$ do
 9.         if $l_i > m_i$ then
10.             return false
11.     return true
```

Algorithm 39 Inclusion Testing with AVX

```
 1. function IS_INCLUDED_AVX(l,m, dim)
 2.     Parameters:
 3.         $l, m \leftarrow$ input matrix
 4.         $dim \leftarrow$ number of variables in program
 5.     $s \leftarrow 2 * dim * (dim + 1)$
 6.     $n \leftarrow 2 * dim$
 7.     $oct\_closure(l, dim)$
 8.     $one \leftarrow avx\_set\_int(1)$
 9.     for $i \leftarrow 0$ to $s/4$ do
10.         $op_1 \leftarrow avx\_load\_double(l_{i*4})$
11.         $op_2 \leftarrow avx\_load\_double(m_{i*4})$
12.         $op_3 \leftarrow avx\_cmp\_leq\_double(op_1, op_2)$
13.         $ip \leftarrow avx\_double\_to\_int(op_3)$
14.         if $(!avx\_nand\_int(ip, one))$ then
15.             return false
16.     return true
```

We vectorize the loop at line 8 using AVX intrinsics. The vectorized pseudo code is shown in Algorithm 39. Here, op_3 contains the result of the $\leq$ comparison between the corresponding elements in l and m. The op_3 is then converted to a 64-bit int and then tested if all of the bits are 1's. The number of times the loop executes depends on the input and so the performance gain from vectorization varies according to the input.

4.4. Equality Testing

Equality operator is used to test if two octagons represent the same set of concrete objects. It involves element wise test for equality between closed matrices. If one of the elements is not equal then the test fails. Algorithm 40 shows the pseudo code for the equality testing operator.

Algorithm 40 Equality Testing

```
 1. function IS_EQUAL(l,m, dim)
 2.     Parameters:
 3.         $l, m \leftarrow$ input matrix
 4.         $dim \leftarrow$ number of variables in program
 5.     $s \leftarrow 2 * dim * (dim + 1)$
 6.     $n \leftarrow 2 * dim$
 7.     $oct\_closure(l, dim)$
 8.     $oct\_closure(m, dim)$
 9.     for $i \leftarrow 0$ to $s$ do
10.         if $l_i ! = m_i$ then
11.             return false
12.     return true
```

Algorithm 41 Equality Testing with AVX

```
 1. function IS_EQUAL_AVX(l,m, dim)
 2.     Parameters:
 3.         $l, m \leftarrow$ input matrix
 4.         $dim \leftarrow$ number of variables in program
 5.     $s \leftarrow 2 * dim * (dim + 1)$
 6.     $n \leftarrow 2 * dim$
 7.     $oct\_closure(l, dim)$
 8.     $oct\_closure(m, dim)$
 9.     $one \leftarrow avx\_set\_int(1)$
10.     for $i \leftarrow 0$ to $s/4$ do
11.         $op_1 \leftarrow avx\_load\_double(l_{i*4})$
12.         $op_2 \leftarrow avx\_load\_double(m_{i*4})$
13.         $op_3 \leftarrow avx\_cmp\_eq\_double(op_1, op_2)$
14.         $ip \leftarrow avx\_double\_to\_int(op_3)$
15.         if $(!avx\_nand\_int(ip, one))$ then
16.             return false
17.     return true
```

We vectorize the loop at line 9 using AVX intrinsics. The vectorized pseudo code is shown in Algorithm 41. The vectorization is the same as for inclusion except the comparison for equality. Again, the number of times the loop executes depends on the input, so the performance gain from vectorization varies according to the input.

4.5. Top

The top operator allocates the highest ordered element in the lattice representing octagons. The top element is represented by a matrix containing only ∞ values and is closed. Algorithm 42 shows the pseudo code for the top operator. The vectorized top operator is shown in Algorithm 43.

Algorithm 42 Top Operator

```
 1. function TOP(m,dim)
 2.     Parameters:
 3.         $m \leftarrow$ input matrix
 4.         $dim \leftarrow$ number of variables
 5.     $n \leftarrow 2 * dim$
 6.     for $i \leftarrow 0$ to $n$ do
 7.         $i_2 \leftarrow (i \vee 1) + 1$
 8.         for $j \leftarrow 0$ to $i_2$ do
 9.             if $i == j$ then
10.                 $m_{i,j} \leftarrow 0$
11.             else
12.                 $m_{i,j} \leftarrow \infty$
```

Algorithm 43 Top Operator with AVX

```
 1. function TOP_AVX(m,dim)
 2.     Parameters:
 3.         $m \leftarrow$ input matrix
 4.         $dim \leftarrow$ number of variables
 5.     $s \leftarrow 2 * dim * (dim + 1)$
 6.     $n \leftarrow 2 * dim$
 7.     $inf \leftarrow avx\_set\_double(\infty)$
 8.     for $i \leftarrow 0$ to $s/4$ do
 9.         $avx\_store\_double(m_{i*4}, inf)$
10.     for $i \leftarrow 0$ to $n$ do
11.         $m_{i,i} \leftarrow 0$
```

The elements on the main diagonal $m_{i,i}$ should be zero which makes vectorization difficult. To make vectorization convenient, we first set all of the elements in the matrix to ∞. We have another loop at line 10 in Algorithm 43 that sets the diagonal elements to 0.

4.6. Is Top

This operator tests if an octagon is the top element. If one of the non diagonal elements in the matrix is not ∞ then it returns false. Algorithm 44 shows the pseudo code for this operator.

Algorithm 44 Is Top

```
1.  function IS_TOP(m, dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $dim \leftarrow$ number of variables in program
5.      $n \leftarrow 2 * dim$
6.      for $i \leftarrow 0$ to $n$ do
7.          $i_2 \leftarrow (i \vee 1) + 1$
8.          for $j \leftarrow 0$ to $i_2$ do
9.              if $i == j$ then
10.                 continue
11.             else if $m_{i,j} == \infty$ then
12.                 continue
13.             else
14.                 return false
15.     return true
```

Algorithm 45 Is Top with AVX

```
1.  function IS_TOP_AVX(m, dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $dim \leftarrow$ number of variables in program
5.      $s \leftarrow 2 * dim * (dim + 1)$
6.      $n \leftarrow 2 * dim$
7.      for $i \leftarrow 0$ to $n$ do
8.          $m_{i,i} \leftarrow \infty$
9.      $flag \leftarrow$ true
10.     $inf \leftarrow avx\_set\_double(\infty)$
11.     $one \leftarrow avx\_set\_int(1)$
12.     for $i \leftarrow 0$ to $s/4$ do
13.         $op_1 \leftarrow avx\_load\_double(m_{i*4})$
14.         $op_2 \leftarrow avx\_cmp\_eq\_double(op_1, inf)$
15.         $ip \leftarrow avx\_double\_to\_int(op_2)$
16.         if $(!avx\_nand\_int(ip, one))$ then
17.             $flag \leftarrow$ false
18.             break
19.     for $i \leftarrow 0$ to $n$ do
20.         $m_{i,i} \leftarrow \infty$
21.     return $flag$
```

The diagonal elements $m_{i,i}$ are set to 0 for all octagons which makes vectorization of equality comparison with ∞ in loop at line 6 difficult. Thus we first set all diagonal elements to ∞ which are afterwards set back to 0. The pseudo code for vectorized operator is shown in Algorithm 45. Like inclusion and equality testing, the number of times the loop executes depends on the input, so the performance gain from vectorization varies according to the input.

4.7. Forget

The forget operator is used to eliminate variables from the octagons. For instance, it is used to model non-deterministic assignments to variables. It sets all the constraints related to the eliminated variable to ∞. The matrix obtained after applying the forget operator is closed. Algorithm 46 shows the pseudo code of the forget operator.

Algorithm 46 Forget Operator

```
1.  function FORGET(m,v,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $v \leftarrow$ the index of variable to be forgotten
5.          $dim \leftarrow$ number of variables in program
6.      $n \leftarrow 2 * dim$
7.      $oct\_closure(m, dim)$
8.      for $i \leftarrow 2v + 2$ to $n$ do
9.          $m_{i,2v} \leftarrow \infty$
10.         $m_{i,2v+1} \leftarrow \infty$
11.     $inf \leftarrow \infty$
12.     for $j \leftarrow 0$ to $2v$ do
13.         $m_{2v,j} \leftarrow \infty$
14.         $m_{2v+1,j} \leftarrow \infty$
15.     $m_{2v,2v} \leftarrow 0$
16.     $m_{2v+1,2v+1} \leftarrow 0$
```

Algorithm 47 Forget Operator with AVX

```
1.  function FORGET_AVX(m,v,dim)
2.      Parameters:
3.          $m \leftarrow$ input matrix
4.          $v \leftarrow$ the index of variable to be forgotten
5.          $dim \leftarrow$ number of variables in program
6.      $n \leftarrow 2 * dim$
7.      $inf \leftarrow avx\_set\_double(\infty)$
8.      $oct\_closure(m, dim)$
9.      for $i \leftarrow 2v + 2$ to $n$ do
10.         $m_{i,2v} \leftarrow \infty$
11.         $m_{i,2v+1} \leftarrow \infty$
12.     $inf \leftarrow \infty$
13.     for $j \leftarrow 0$ to $2v/4$ do
14.         $avx\_store\_double(m_{2v,j*4}, inf)$
15.         $avx\_store\_double(m_{2v+1,j*4}, inf)$
16.     $m_{2v,2v} \leftarrow 0$
17.     $m_{2v+1,2v+1} \leftarrow 0$
```

The loop at line 8 can be easily vectorized as elements are accessed row-wise. The elements for the loop at line 12 are accessed column-wise. Thus, this loop cannot be vectorized. There is no gain in memory performance by storing columns in array as the operation is linear. The pseudo code for the vectorized forget operator is shown in Algorithm 47.

5

Evaluation

We next present an implementation of dense and sparse libraries for static analysis with the Octagon abstract domain. We evaluate the performance of various closure and incremental closure algorithms on real-world and synthetic benchmarks. We then compare the performance of our dense and sparse libraries against APRON. To compare, we measure the time that the analysis spends in the Octagon domain. We also measure the individual runtime of Octagon operators. Some operators are part of others, for example *join* includes *closure*. When we measure the runtime of such operators, we only measure the runtime for the portion which does not belong to the included operator. Hence, for *Join*, we only consider the time spent in computing the maximum of matrix elements in a quadratic loop. The time for *closure* in *join* is included in the total *closure time*.

5.1. Library Implementation

In this section we describe the implementation of our sparse and dense libraries for static analysis using the Octagon domain.

5.1.1. Octagons

The octagons are represented in the same way as in APRON. We store two matrices, *closed* and *mat*. *Closed* stores the closed version of matrix *mat*. This allows us to skip performing closure, if the closed version is already available. If an operation destroys closeness of the matrix, then *closed* is set to NULL. Similarly, if an operation makes matrix closed then *mat* is set to NULL.

When the closure is explicitly called, we compute the *closed* matrix and also keep the original *mat* matrix.

```
struct opt_oct_t{
        double *mat;
        double *closed;
        int dim;
}
```

An alternative design for octagons could be to store a boolean indicating if the matrix is closed. However, the operator *widening* requires non closed matrix for convergence. Thus, we store two matrices for an octagon.

5.1.2. Constraints

We use the APRON data structures for representing linear and non-linear constraints. We use the generic linearization algorithms provided by APRON for converting non linear constraints to linear constraints.

5.1.3. Operators

There are separate sparse and dense implementations for closure and incremental closure. We keep a flag for compiling the library to either use sparse or dense implementation. It is trivial to add a field *sparsity* to the Octagon representation that keeps track of the sparsity of matrices and switches between dense and sparse algorithms depending on the sparsity of matrices. For all our benchmarks, we found that the sparse library is always faster than the dense and computation of *sparsity* carries an overhead, so we do not support it currently. All the other operators are optimized for memory performance and most of them use explicit vectorization using AVX intrinsics.

We do not have a sparse implementation for other operators because it will require having either an explicit sparse data structure for octagons or maintaining additional data structure inside octagons to store the locations of finite values. Choosing the first option will slow down the closure like in the case for the Johnson implementation and will incur extra overhead in case we have to switch between dense and sparse (see section 5.3) while with the second option, the copy operation will become more expensive as there will be more bytes to copy. For our benchmarks, copy is one of the most expensive operators for the sparse library so we discarded the second option (see section 5.6).

5.2. Experimental setup

All the experiments were carried out on 3.5 GHz Intel Quad Core i7-4771 Haswell CPU. The sizes of the L1, L2 and L3 caches were 256 KB, 1024 KB and 8192 KB respectively. The size of the main memory was 16GB. Turbo boost was disabled for consistency of measurements. The compiler used was gcc 4.8 with flags *O3*, *-m64* and *-march=native*. Compiler auto vectorization was enabled for APRON and disabled for our libraries.

5.3. Closure Evaluation

We evaluate the performance of implementations of various closure algorithms on synthetic and real inputs. For synthetic (random) inputs, we generate a non sparse random matrix of size 1024×1024 without any ∞ values. The real input is the trace of closure operation extracted by running DPS analysis on the CRYPT benchmark (see section 6.6.1). The generated matrices are very sparse and the maximum size is 474×474. The experiments are carried out using 8 byte double precision.

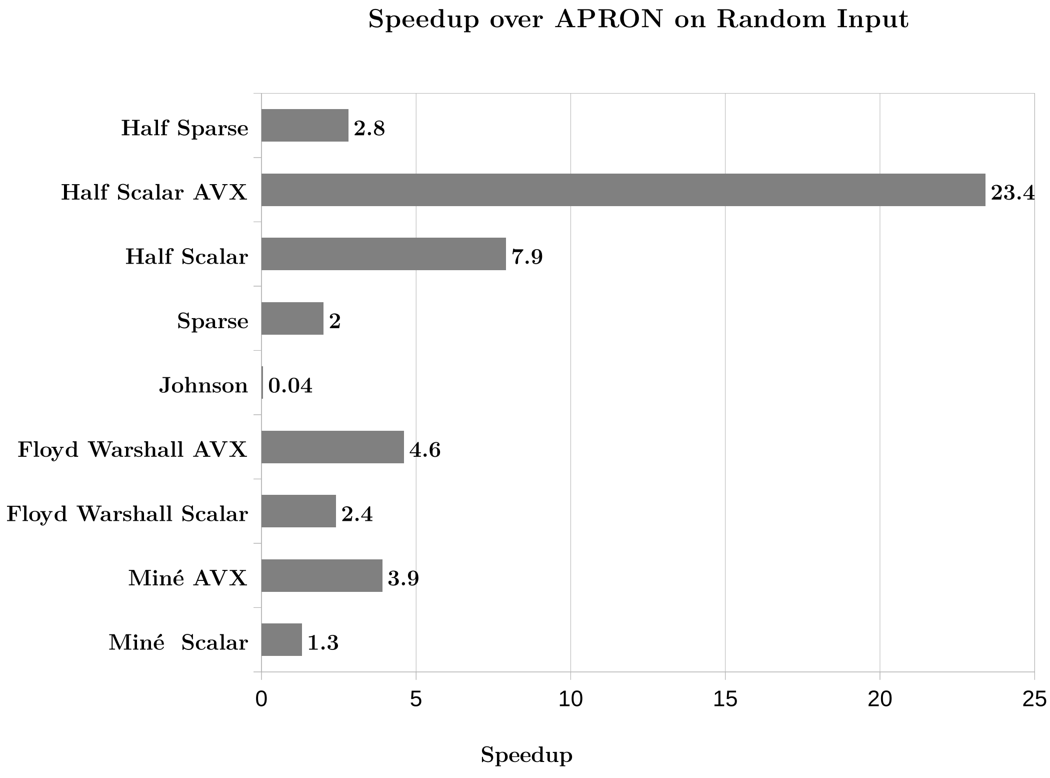

Figure 5.1.: Comparison of Closure algorithms on random input

Figure 5.1 shows the speedup over Apron for the closure algorithms on random input. Memory optimizations and halving the number of operations for algorithms working with half matrix results in large speedups. The half scalar is 7.9 and half AVX is 23.4 times faster than APRON. The sparse implementations are also faster than APRON even for random inputs. The sparse is 2 and half sparse is 2.8 times faster than APRON. Thus, in case the matrices in the analysis become dense from sparse, we still remain faster than APRON. Johnson's [11] algorithm is considerably slower than APRON. This is due to the complex data structures like priority queue used by the Dijkstra algorithm [9] . This highlights the need of choosing the "correct" data structure for performance.

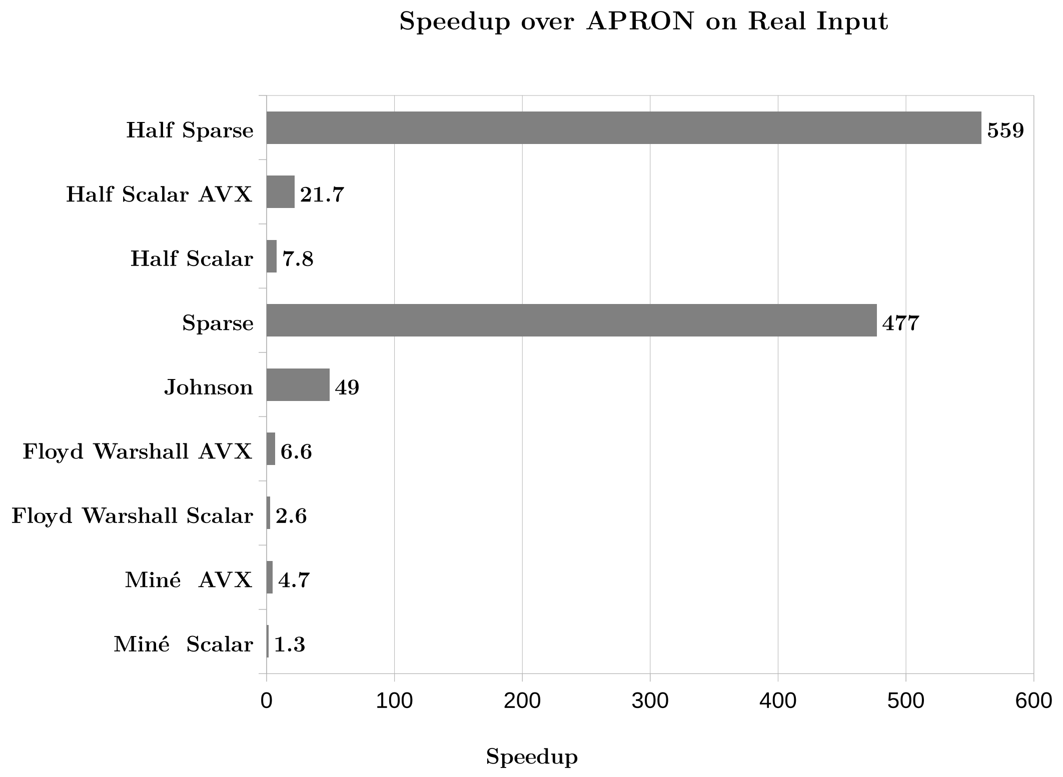

Figure 5.2.: Comparison of Closure algorithms on real input

Figure 5.2 shows the speedups over Apron for closure algorithms on real input. Johnson's algorithm exploits the sparsity of matrices and is 49 times faster than APRON. However, it is outperformed by our sparse algorithms. The sparse is 477 times and half sparse is 559 times faster than APRON. The non sparse algorithms provide similar speed up for real input as for random input.

5.4. Incremental Closure Evaluation

We evaluate the performance of incremental closure algorithms on random and real benchmarks. The random input is a random matrix of size 1024×1024. The index v is set to 0 that is the hardest for half matrix algorithms as elements are accessed column-wise. The real input is the trace of incremental closure operation extracted by running DPS analysis on CRYPT benchmark. The generated matrices are very sparse and maximum size is 474×474. The experiments are carried out using 8 byte double precision.

Figure 5.3 shows the speedup over APRON for incremental closure algorithms on random input. The half dense AVX is 5.5, half dense scalar is 4.2 and half sparse is 3 times faster than APRON. As the incremental closure is quadratic, the speedups achieved are smaller than for closure. Our sparse implementation is again faster than APRON even for random input. Thus, we remain faster than APRON even if matrices become dense as the analysis proceeds.

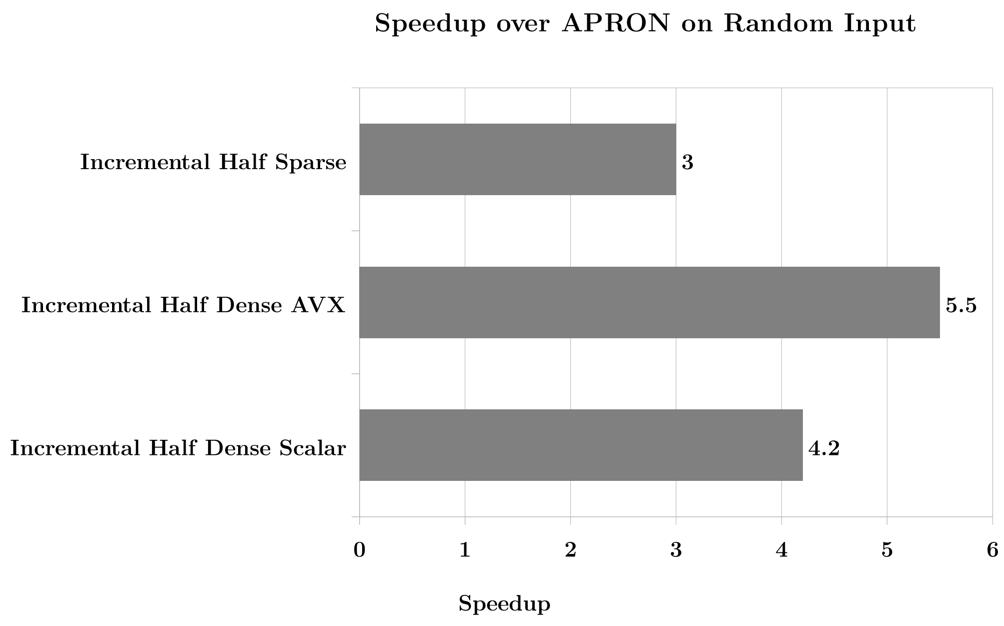

Figure 5.3.: Comparison of Incremental Closure algorithms on random input

Figure 5.4 shows the speedup over APRON for incremental closure algorithms on real input. The sparse implementation exploits sparsity well and achieves a speedup of 60 over APRON. The dense implementations achieve similar speedups for both real and random inputs.

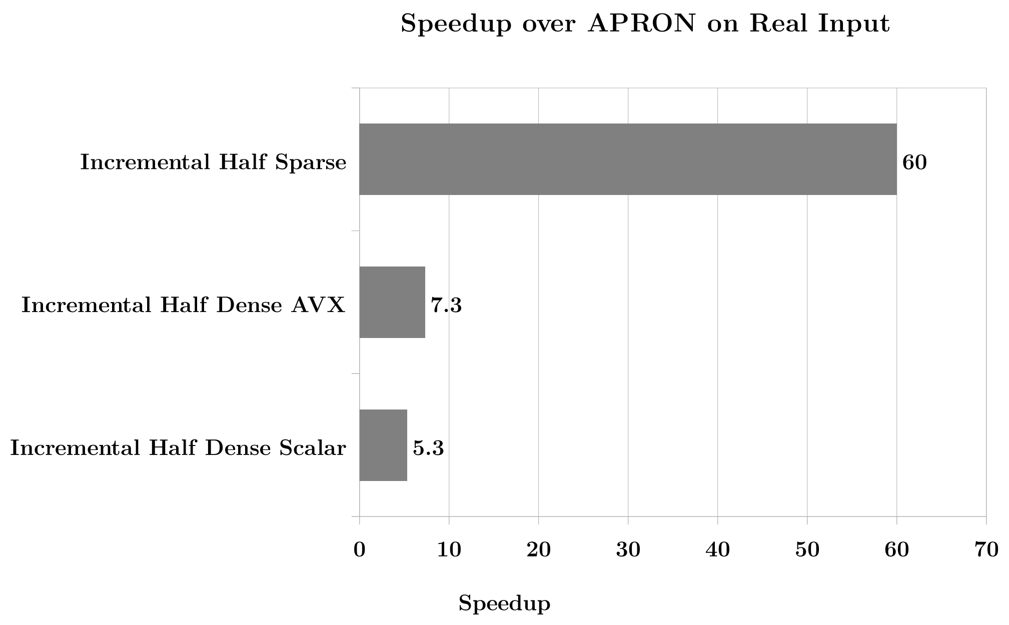

Figure 5.4.: Comparison of Incremental Closure algorithms on real input

5.5. Benchmarks

We used two real world program analysis engines for comparing the performance of our sparse and dense library against APRON. The first analysis is called DPS [28] and is developed at the Software Reliability Lab in ETH Zurich. The analysis statically introduces synchronization for a potentially non-deterministic parallel program. This removes the non-determinism and makes the program deterministic. The analysis is implemented in Java and uses the Soot [31] framework. There are six input benchmarks [2] on which the analysis runs. The second analysis is DIZY [27] developed at Technion, Israel. It computes semantic differences between a program and a patched version of the same program. The analysis is written in C++ and uses LLVM [20] and CLANG [1]. We chose 6 benchmarks which are reasonably large for DIZY.

5.6. Results

In this section we compare the performance of our libraries against APRON on benchmarks in the two analyses.

5.6.1. DPS

The six input benchmarks used by the DPS analysis are described in Table 5.1. The maximum number of variables that are present during analysis are also shown for each benchmark. DPS analysis transforms the input program into Static Single Assignment(SSA) intermediate representation. The largest benchmark is CRYPT and contains a maximum of 237 variables in SSA form. We run the DPS analysis for two different values of the widening threshold. The widening threshold controls when widening is applied and thus affects the runtime of the analysis. The analysis is carried out in double precision.

Program	Description	Max Number of Variables
CRYPT	IDEA Encryption	237
MOLDYN	Molecular Dynamics Simulation	67
LUFACT	LU Factorization	31
SOR	Successive Over-Relaxation	54
MATMULT	Sparse Matrix Multiplication	24
SERIES	Fourier Coefficient Analysis	21

Table 5.1.: Description of Benchmarks in DPS Analysis

Figure 5.5 compares the runtime in CPU cycles of various Octagon operators on CRYPT benchmark for widening threshold=9. Closure accounts for more than 95% of the analysis time for

APRON. The SSA representation creates new variable for every assignment. Thus, variables are not very interrelated which results in matrices being very sparse. Thus, exploiting sparsity yields huge speedup for closure. The dense closure is also very fast compared to APRON closure. The second most time consuming operator for APRON is meet with linear constraint (see appendix A). The bottleneck for this operator is computation of incremental closure. Again, exploiting sparsity in computation of incremental closure yields significant speedup.

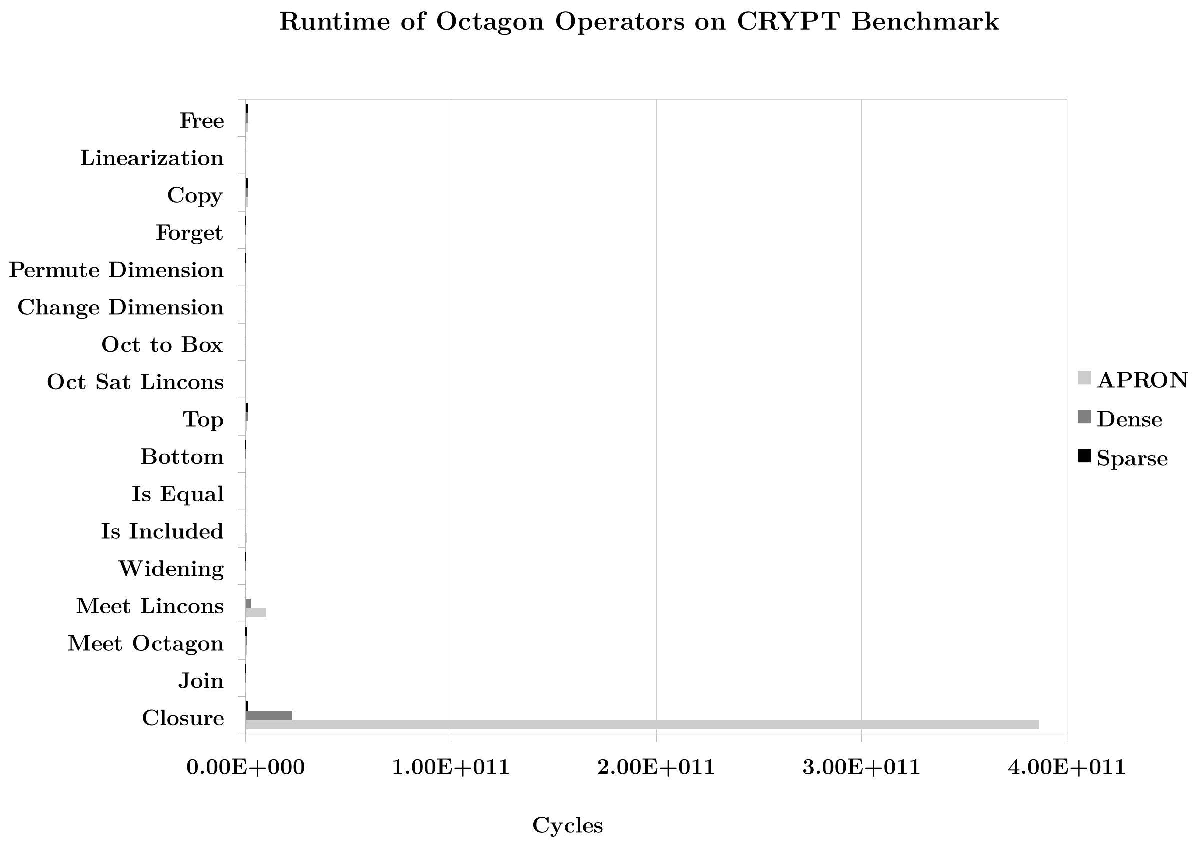

Figure 5.5.: Octagon operators on CRYPT benchmark with widening threshold = 9

Figure 5.6 compares the runtime in cycles of various Octagon operators on the SOR benchmark for widening threshold=9. SOR contains fewer number of variables than CRYPT. Meet with Linear Constraint is the most expensive operation for APRON. This shows that different benchmarks can have different runtime profile depending on the structure of the program. Due to the sparse nature of matrices, the sparse library yields speedup on meet with linear constraint. The closure is the next expensive operator. As the matrices are smaller, the speedup for closure here is smaller compared to CRYPT. Copy and Linearization account for significant portion of total runtime for APRON. These operators are memory bound and are not optimized in our libraries. For Sparse library, Copy and Linearization are the most expensive operators. Thus, our overall speedup is reduced.

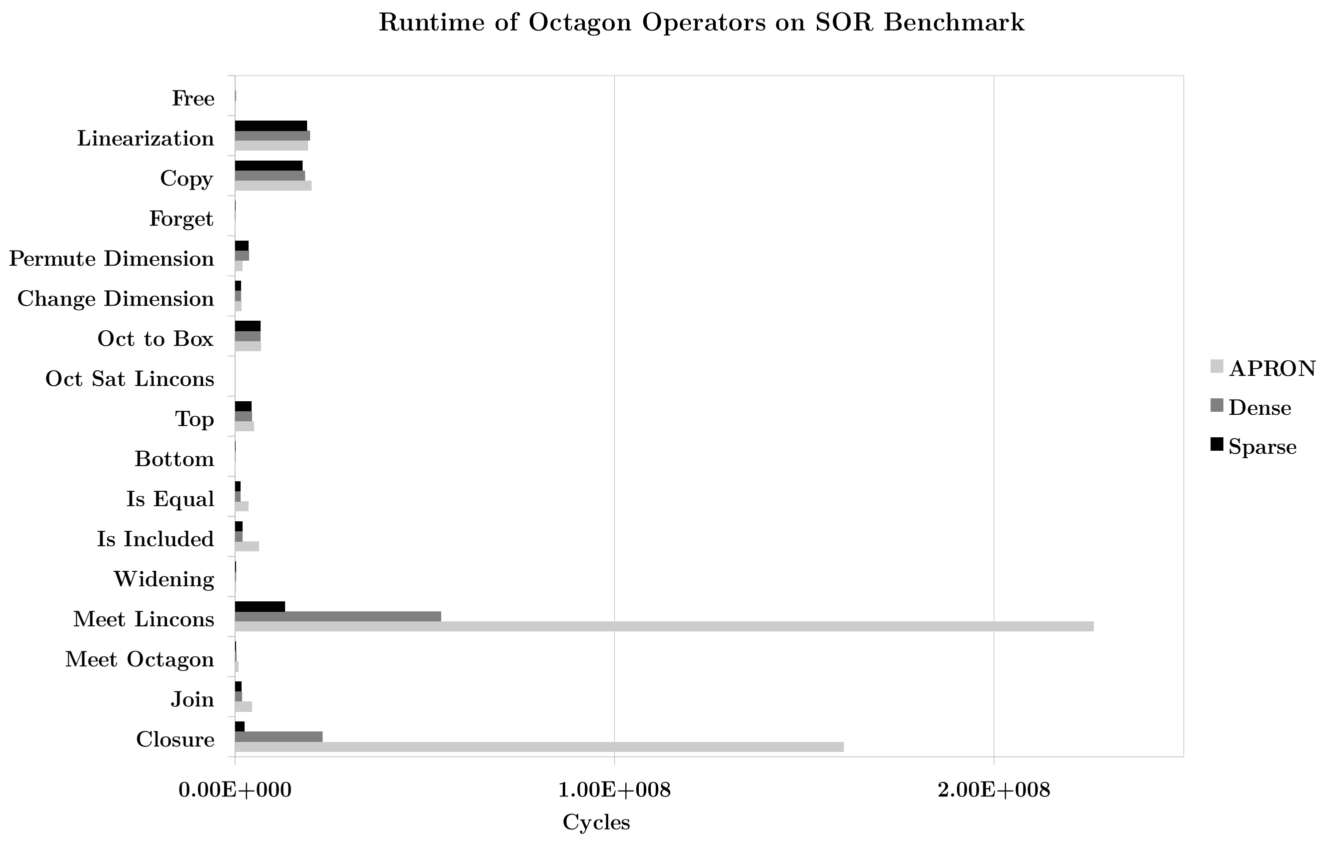

Figure 5.6.: Octagon operators on SOR benchmark with widening threshold = 9

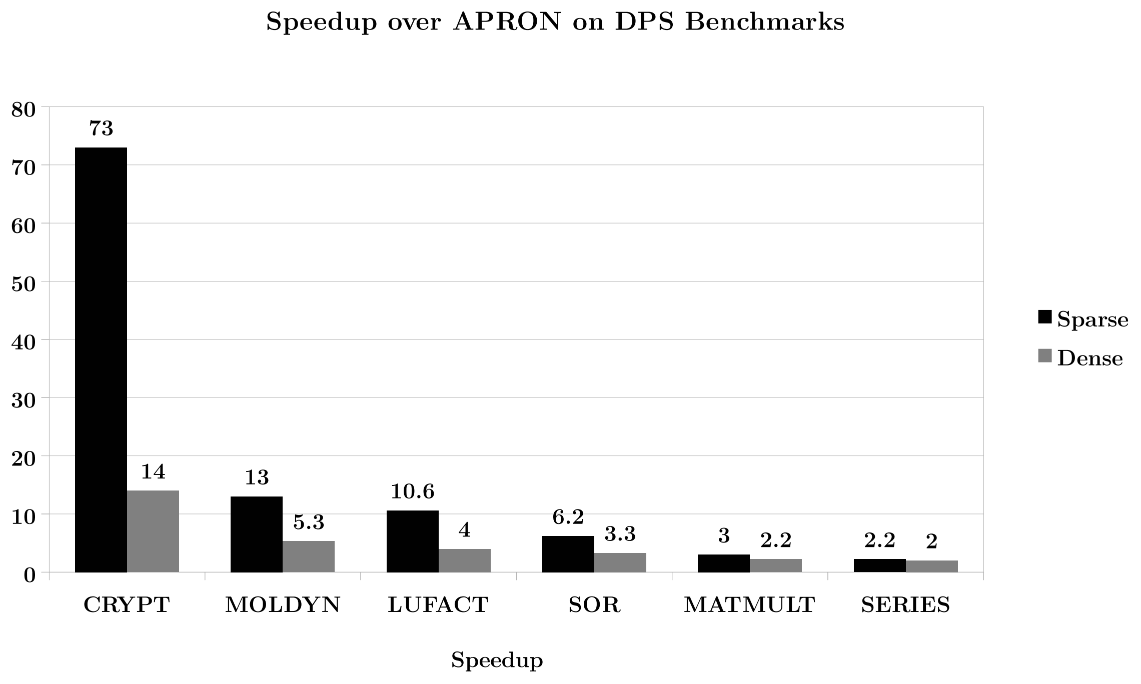

Figure 5.7.: Speedup for dense and sparse libraries over APRON on DPS Benchmarks with widening threshold = 9

Figure 5.7 shows the speedup of overall Octagon analysis over APRON for our dense and sparse libraries on benchmarks in DPS Analysis with widening threshold=9. For CRYPT, the sparse library is 73 and dense library is 14 times faster than APRON. Significant speedup is also achieved for MOLDYN, LUFACT and SOR benchmarks. The performance gain decreases as the number of variables in the program decreases and the analysis time decreases for MATMULT and SERIES. For all 0f the benchmarks, the sparse library is faster than dense.

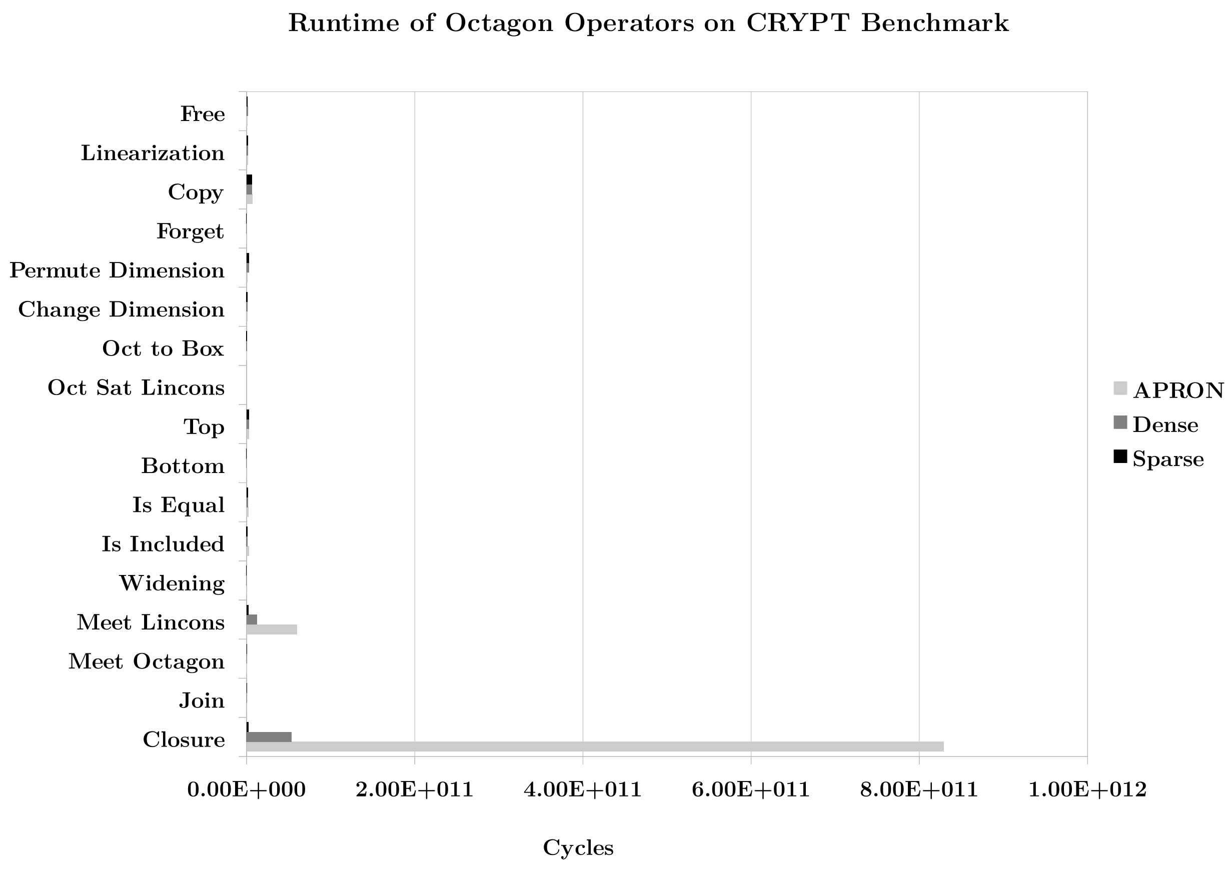

Figure 5.8.: Octagon operators on CRYPT benchmark with widening threshold = 101

Figure 5.8 compares the runtimes in CPU cycles of various Octagon operators on the CRYPT benchmark for widening threshold=101. Closure and meet with linear constraints are again the most expensive operators for APRON. Sparse and dense libraries are both optimized on these operators and provide significant speedups. However, as the analysis time increases due to a larger widening threshold, there are more copy operations. In fact, copy is the most expensive operator for the sparse library. This reduces our overall speedup for both sparse and dense libraries.

Figure 5.9 compares the runtime in cycles of various Octagon operators on SOR for widening threshold=101. Meet with linear constraints and closure are the two most expensive operators for APRON. Sparse and Dense libraries both provide significant speedup on these operators. However like in the case for CRYPT, the contribution of copy operator to the total runtime increases which reduces overall speedup.

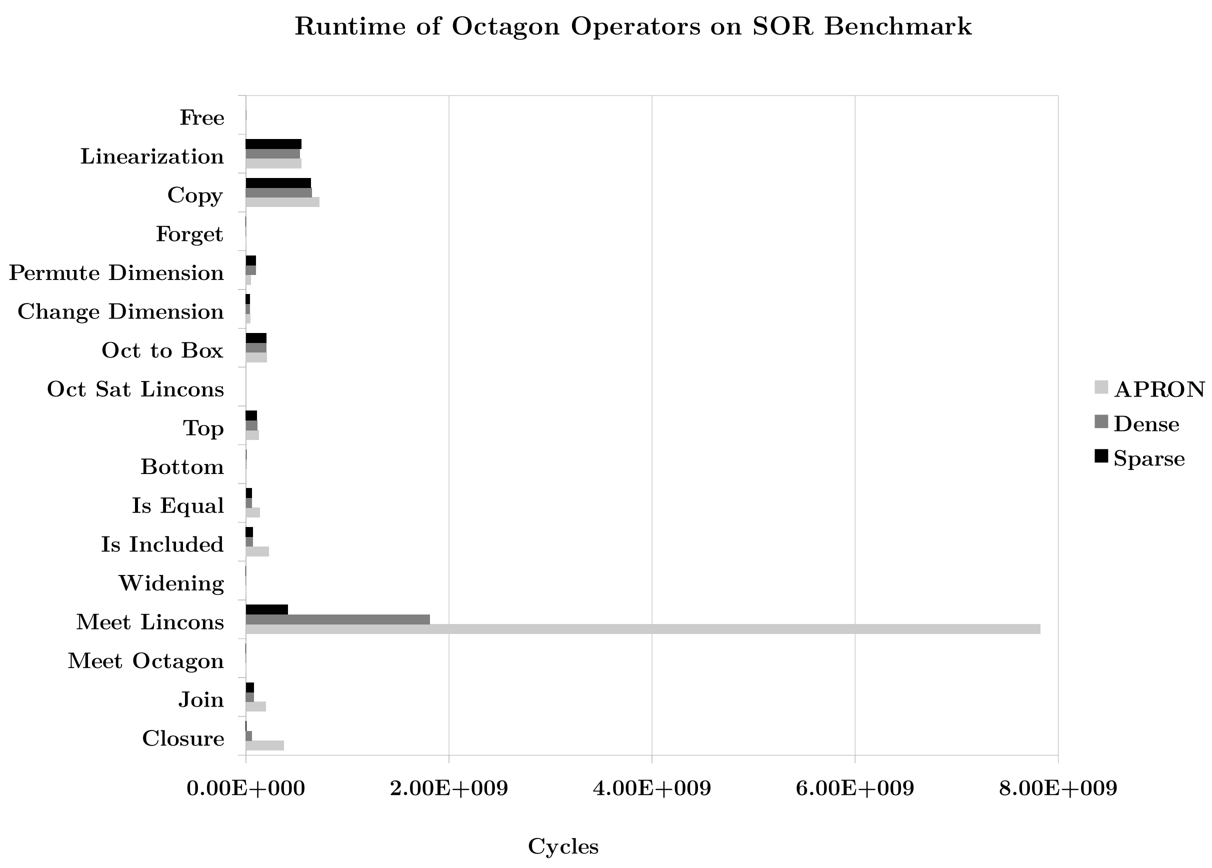

Figure 5.9.: Octagon operators on SOR benchmark with widening threshold = 101

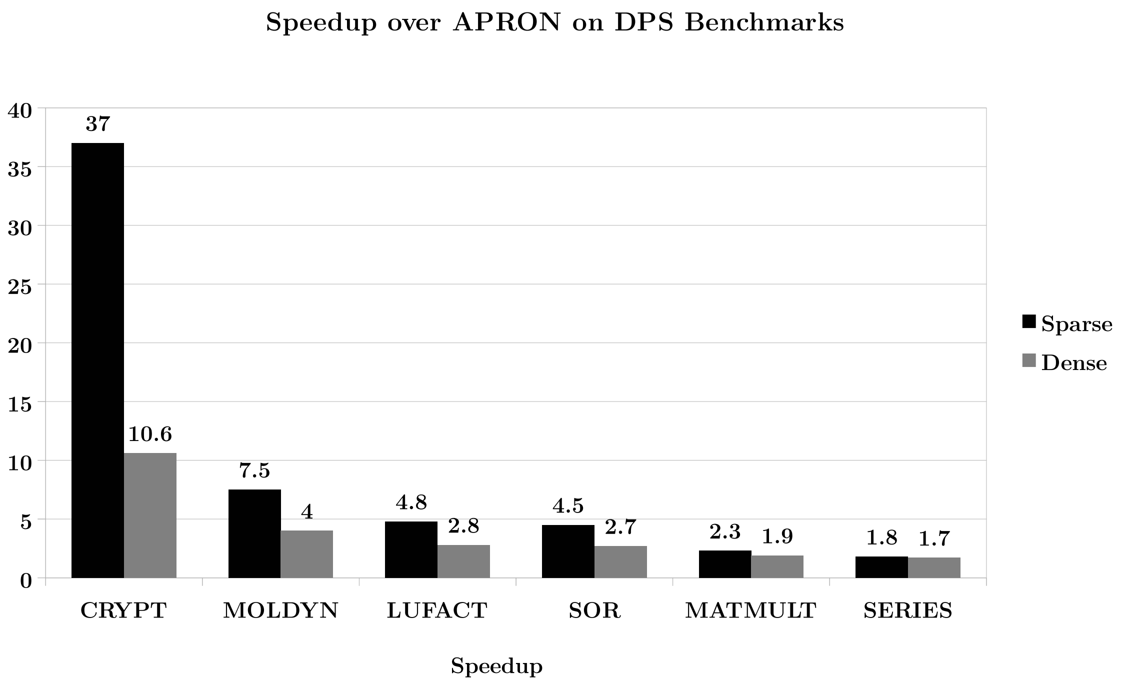

Figure 5.10.: Speedup for dense and sparse libraries over APRON on DPS Benchmarks with widening threshold = 101

Figure 5.10 shows the speedup of overall analysis over APRON for our dense and sparse libraries on benchmarks in DPS analysis. Highest speedup is achieved for CRYPT. However, compared with widening threshold=9, the speedup is reduced because of more copy operations. The decrease in speedup is also observed for all other programs. The sparse library is faster than dense on all benchmarks .

5.6.2. DIZY

Most of the benchmarks provided with the DIZY analysis have a very small number of variables and runtime. We picked the six most time consuming benchmarks from the analysis in order to achieve meaningful speedups. The benchmarks are shown in table 5.2. The table also shows the maximum number of variables that occur during the analysis.

Program	Description	Max Number of Variables
LINUX_FULL	Linux Kernel Function	78
SEQ	Print Arithmetic Sequence	35
FIREFOX	Mozilla Firefox Function	24
AEG	Array Assignment	10
MD5SUM	MD5 Hash	18
MD5SUM_LOOP	Modified MD5 Hash	10

Table 5.2.: Description of Benchmarks in DIZY Analysis

The benchmarks of the DIZY analysis contain a smaller number of variables compared to the DPS benchmarks. This is due to the fact that DIZY runs directly on the original program without transforming it into SSA-like intermediate representation. This also reduces the sparsity of the matrices as variables in program are much more interrelated than variables in SSA representation. The analysis uses partitioning technique to reduce the runtime of analysis. We run the analysis in two configurations: with partitioning and without partitioning. The analysis originally used 32 byte MPQ numbers whereas our library uses 8 byte doubles. In order to have a fair comparison we changed the analysis so that it also uses 8 byte doubles. The largest benchmark is LINUX_FULL and contains a maximum of 74 variables.

Figure 5.11 compares the runtime in CPU cycles of various Octagon operators on the LINUX_FULL benchmark without partitioning. Closure and copy are the two most expensive operators for APRON. Sparse and dense libraries provide huge speedup for closure. However, copies are expensive for these libraries and reduce the overall speedup.

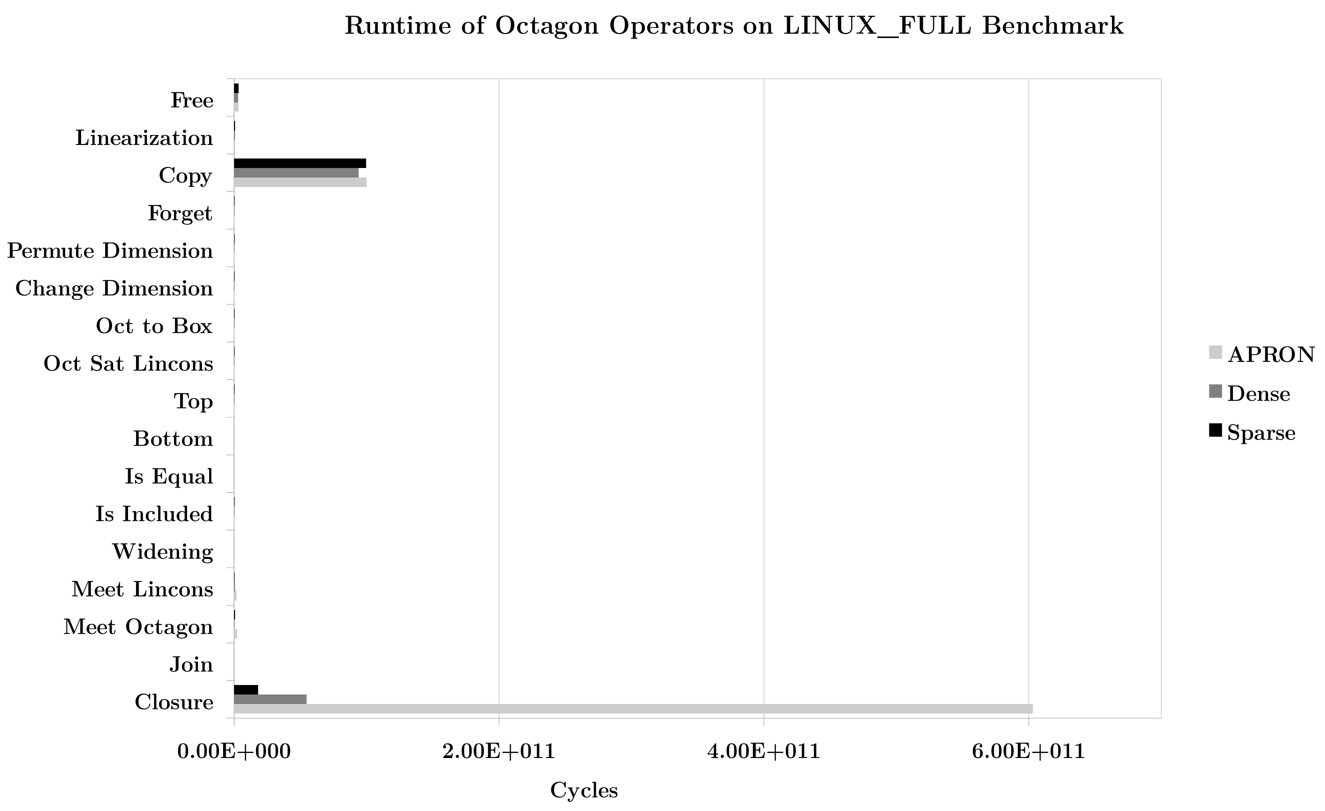

Figure 5.11.: Octagon operators on LINUX_FULL benchmark with no partitioning

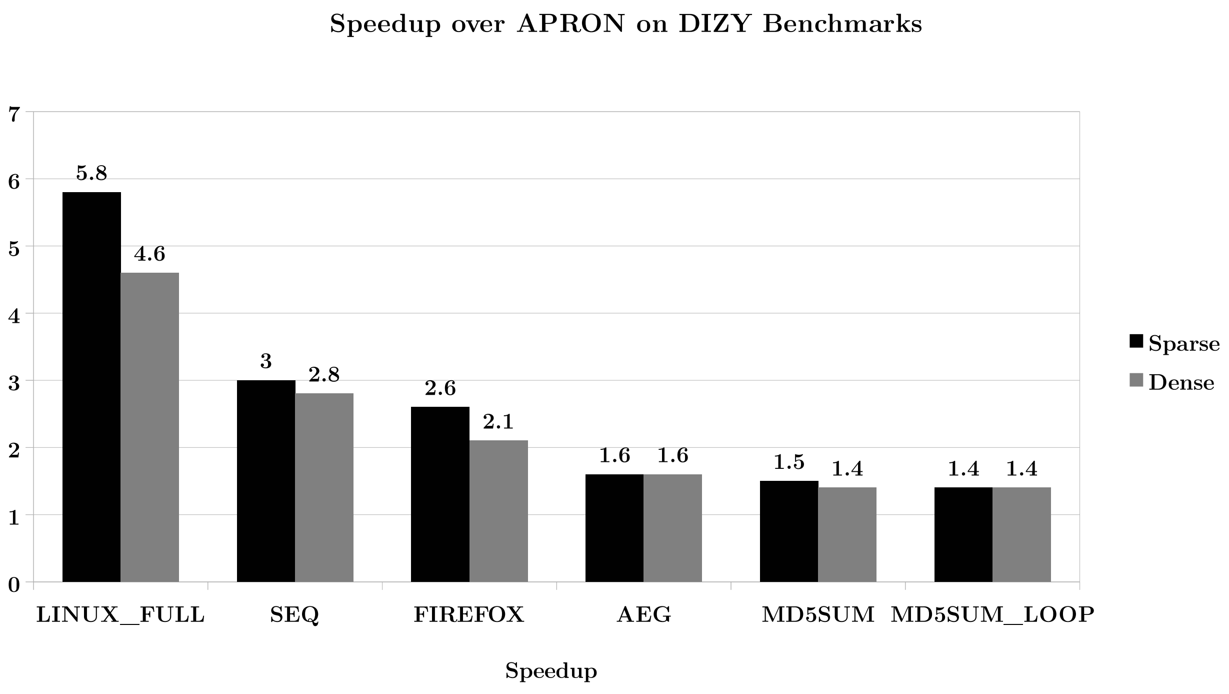

Figure 5.12.: Speedup for dense and sparse libraries over APRON on DIZY Benchmarks with no partitioning

Figure 5.12 shows the speedup of overall analysis over APRON for the different benchmarks in the DIZY analysis without partitioning. The highest speedup is achieved for the LINUX_FULL benchmark. Sparse and dense libraries are respectively 5.8 and 4.6 times faster than APRON. There is a significant speedup for the SEQ and FIREFOX benchmarks, while for the rest of the benchmarks, the speedups are smaller. The sparse library is faster than dense for all benchmarks.

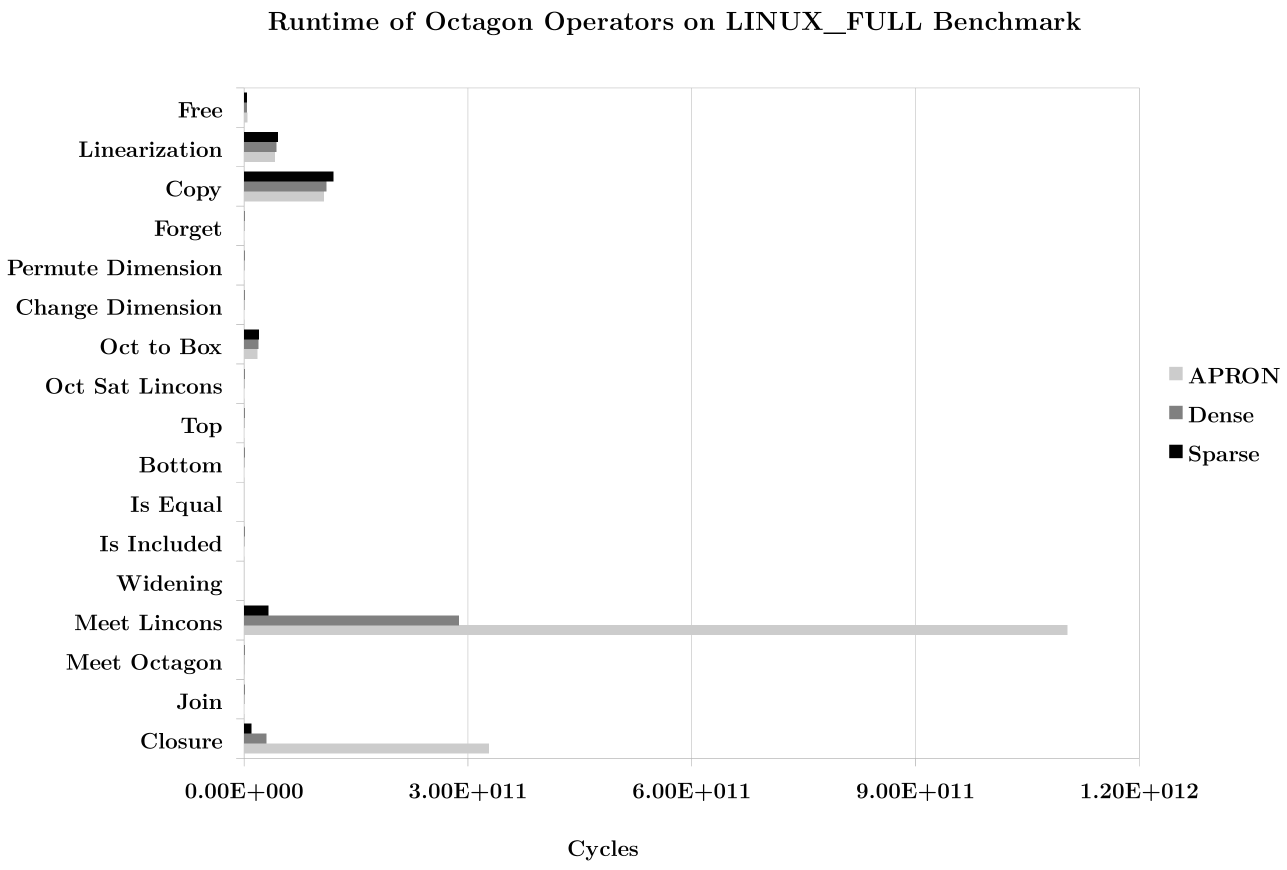

Figure 5.13.: Octagon operators on LINUX_FULL benchmark with partitioning

Figure 5.13 compares the runtime in CPU cycles of Octagon operators on the LINUX_FULL benchmark with partitioning. There is very different distribution of runtime among Octagon operators with partitioning. This shows that the runtime profile also depends on how the analysis is carried out. Meet with linear constraint is the most expensive operator for APRON. Both libraries provide huge speedup for this operator. Closure is the next expensive operator and is optimized for performance in our libraries. Copy and Linearization are also expensive and reduce the overall speedup.

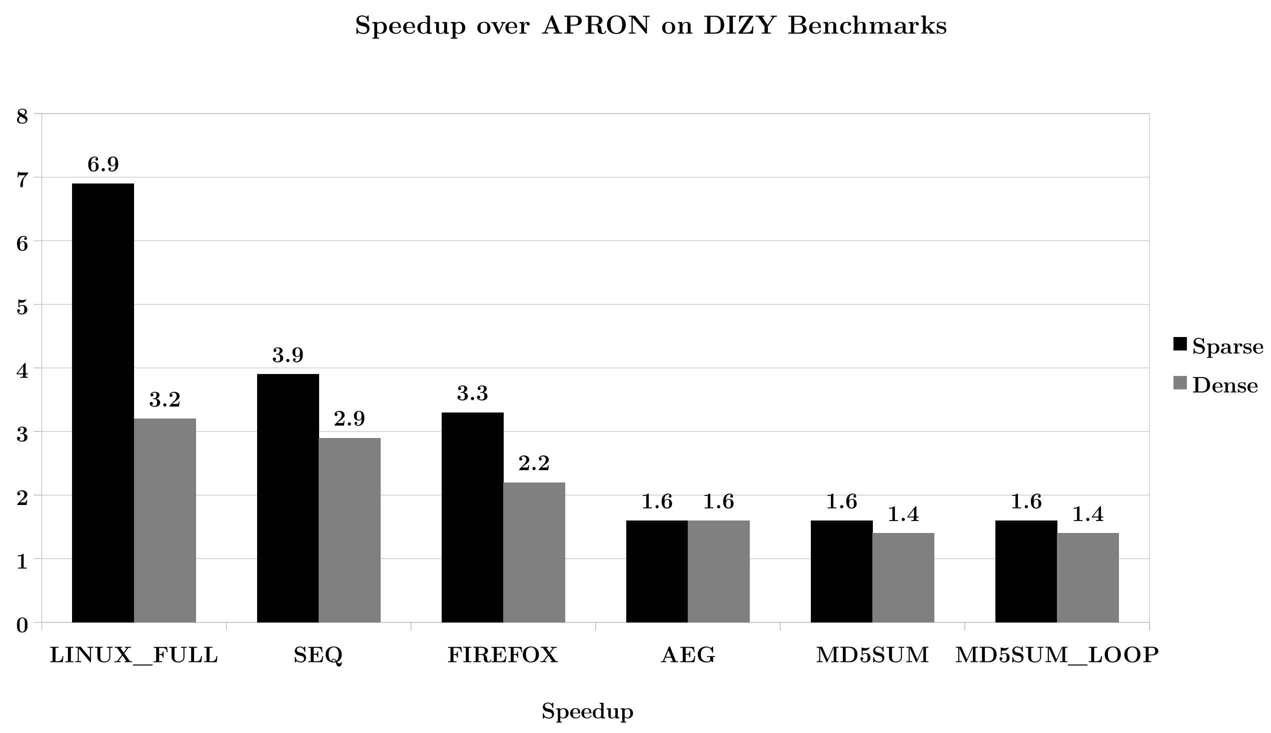

Figure 5.14.: Speedup for dense and sparse libraries over APRON on DIZY Benchmarks with partitioning

Figure 5.14 shows the speedup of overall analysis over APRON for different benchmarks in DIZY analysis with partitioning. Again, highest speedup is achieved for LINUX_FULL benchmark. Sparse is 6.9 and dense is 3.2 times faster than APRON. The speedup is reduced for dense library because most of the time is spent in computing meet with linear constraint compared with closure earlier. The dense library provides larger speedup for closure than for incremental closure. Significant speedup is achieved for SEQ and FIREFOX benchmarks. The sparse library is again faster for all benchmarks.

6

Conclusion and Future Work

In this thesis, we introduced new algorithms which significantly speed up the operators of the Octagon abstract domain. Our new algorithms take advantage of memory optimizations, vector hardware instructions, and sparsity and structure of matrices. We achieved our speedups mainly by optimizing the (key) closure operator, which is the most expensive operator in the Octagon domain.

There are other related domains that are more expensive than Octagon. We would like to optimize these in future work as well.

The Two Variables Per Inequality (TVPI)[30] domain encodes linear relationships of the form $a_1x_i + a_2x_j \leq c$. Even though like the Octagon, each constraint for TVPI involves at most two variables, the number of inequalities between two variables can be arbitrarily large. It also defines a closure operation[18] which is more expensive than Octagon closure. The join is computed by computing the convex hull of input inequalities and is very expensive. The forget operator involves variable elimination by Fourier-Motzkin [14] which can generate an exponential number of inequalities.

The Octahedron domain[10] encodes relationships of the form $\pm x_i \leq c$ and is more precise than Octagon. The domain is implemented using Octahedra Decision Diagrams (ODD). It defines saturation operation which is similar to computing closure. The saturation operator produces 3^n number of inequalities. Other operators like join use saturation as suboperation thus their cost is also exponential.

The Polyhedra domain [13][8] has exponential time and space complexity. There are two representations for polyhedron. It can either be represented as the conjunction of a finite set of linear constraints or as a finite collection of vertices or rays. The former representation is called constraint while the latter is called frame representation. Some operators (e.g., meet and test) can be implemented more efficiently on the constraint representation, while others (e.g., forget and join) can be performed more efficiently on the frame representation. Thus, it is often necessary to convert from one representation to the other. The dual conversions are performed using the Chernikova algorithm [21] which can produce an output that is exponential in the size of the input.

Besides numerical domains, there are domains (e.g. Three Valued Logic Analysis (TVLA) [29]) for verifying properties related to heap like memory leaks, null pointer errors, dangling pointers etc. Optimizing such non numerical domains is challenging and is part of our future work.

It must be noted that although we improved the performance of numerical domains, the overall static analysis may still have other bottlenecks due to other components such as for instance the front end. In the future, we would also like to optimizing these components.

Our long term goal is to develop a code generator that produces optimized code for static analyzers targeting particular architectures by exploiting memory optimization, hardware vector instructions and sparsity. Towards this, we plan to develop a Domain Specific Language (DSL) [24] for static analysis. The user will write his analysis in our DSL and will specify the numerical domain as a parameter. Our synthesizer will then generate code optimized for a particular hardware and abstract domain.

Bibliography

[1] Clang: A C Language family Frontend for LLVM http://clang.llvm.org/.

[2] Habanero Multicore Software Research Project. http://habanero.rice.edu/hj.

[3] http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[4] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[5] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Weakly-relational Shapes for Numeric Abstractions: Improved Algorithms and Proofs of Correctness. *Form. Methods Syst. Des.*, 35(3):279–323, December 2009.

[6] F. Banterle and R. Giacobazzi. A Fast Implementation of the Octagon Abstract Domain on Graphics Hardware. In G. Filé and H.R. Nielson, editors, *Proc. of The 14th International Static Analysis Symposium, SAS'07*, volume 4634 of *Lecture Notes in Computer Science*, pages 315–335. Springer-Verlag, 2007.

[7] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 196–207, New York, NY, USA, 2003. ACM.

[8] Liqian Chen, Antoine Miné, and Patrick Cousot. A Sound Floating-Point Polyhedra Abstract Domain. In *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 3–18. Springer Berlin Heidelberg, 2008.

[9] Mo Chen, Rezaul Alam Chowdhury, Vijaya Ramachandran, David Lan Roche, and Lingling Tong. Priority Queues and Dijkstra's Algorithm, 2007.

[10] Robert Clariso and Jordi Cortadella. The Octahedron Abstract Domain. In *In Static Analysis Symposium (2004*, pages 312–327. Springer-Verlag, 2004.

[11] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[12] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[13] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[14] George B Dantzig and B Curtis Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory, Series A*, 14(3):288 – 297, 1973.

[15] Robert W. Floyd. Algorithm 97: Shortest Path. *Commun. ACM*, 5(6):345–, June 1962.

[16] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[17] Sung-Chul Han, Franz Franchetti, and Markus Püschel. Program Generation for the All-pairs Shortest Path Problem. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 222–232, New York, NY, USA, 2006. ACM.

[18] Jacob M. Howe and Andy King. Closure Algorithms for Domains with Two Variables Per Inequality. Technical report, 2009.

[19] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer Berlin Heidelberg, 2009.

[20] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[21] Hervé Le Verge. A Note on Chernikova's algorithm. Rapport de recherche RR-1662, INRIA, 1992.

[22] Francesco Logozzo and Manuel Fahndrich. Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses. *Science of Computer Programming*, 75(9):796 – 807, 2010.

[23] Laurent Mauborgne. ASTRÉE: Verification of Absence of Run-Time Error. In *Building the information Society (18th IFIP World Computer Congress)*, pages 384–392. The International Federation for Information Processing, Aug 2004.

[24] M. Mernik, J. Heering, A.M. Sloane, Marjan Mernik, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages, 2003.

[25] Antoine Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Proceedings of the Second Symposium on Programs As Data Objects*, PADO '01, pages 155–172, London, UK, UK, 2001. Springer-Verlag.

[26] Antoine Miné. The Octagon Abstract Domain. *Higher Order Symbol. Comput.*, 19(1):31–100, March 2006.

[27] Nimrod Partush and Eran Yahav. Abstract Semantic Differencing for Numerical Programs. In *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 238–258. Springer Berlin Heidelberg, 2013.

[28] Veselin Raychev, Martin Vechev, and Eran Yahav. Automatic Synthesis of Deterministic Concurrency. In *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 283–303. Springer Berlin Heidelberg, 2013.

[29] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[30] Axel Simon, Andy King, and Jacob M. Howe. Two Variables Per Linear Inequality as an Abstract Domain. In *Logic-based Program Synthesis and Transformation, volume 2664 of LNCS*, pages 71–89. Springer, 2003.

[31] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - A Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[32] Arnaud Venet and Guillaume Brat. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 231–242, New York, NY, USA, 2004. ACM.

[33] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. A Blocked All-pairs Shortest-paths Algorithm. *J. Exp. Algorithmics*, 8, December 2003.

A

Appendix

A.1. Closure Algorithms

A.1.1. Floyd Warshall

Bagnara et al. [5] showed that applying Floyd Warshall[15][17][33] all pairs shortest path directly also ensures property 1 and 2 of octagon closure. Thus, Floyd-Warshall can be applied directly. The pseudo-code for computing octagon closure using Floyd-Warshall is shown in Algorithm 48. Han et al. [17] show how to optimize Floyd-Warshall for performance using memory optimizations and vectorization. They take advantage of the fact that the values in *k*-th row and column do not change during *k*-th iteration of outermost loop. This allows them to efficiently perform tiling which improves memory and vectorization performance.

Algorithm 48 Closure With Floyd Warshall

```
1. function CLOSURE(m,dim)
2.     $m \leftarrow$ input matrix
3.     $dim \leftarrow$ number of variables in program
4.     for $k \leftarrow 0$ to $2 * dim$ do
5.         for $i \leftarrow 0$ to $2 * dim$ do
6.             for $j \leftarrow 0$ to $2 * dim$ do
7.                 $m_{i,j} \leftarrow min(m_{i,j}, m_{i,k} + m_{k,j})$
8.     return $Strengthening(m, dim)$
```

A.1.2. Johnson's Closure

Johnson's algorithm[11] can also be used to compute all pairs shorts path. It works on a graph representation and has lower complexity than Floyd-Warshall for sparse graphs. If V be the number of nodes in graph and E be the number of edges then the complexity of Johnson's algorithm is $O\big(VE+V^2logV\big)$. If E is of order of V then complexity becomes $O\big(V^2logV\big)$. The pseudo code for computing octagon closure using Johnson's algorithm is shown in Algorithm 49.

Algorithm 49 Closure with Johnson's algorithm

```
1. function CLOSURE(m,g,dim)
2.     $n$ ←2*dim
3.     Add a vertex $q$ to the graph and connect it to all $n$ nodes using zero weight edges.
4.     Use Bellman-Ford to compute shortest path $d(v)$ from $q$ to each vertex $v$, terminate if negative cycle is detected
5.     Reweigh the edges in original graph, $\forall i, j, m_{i,j} \leftarrow m_{i,j} + d(i) - d(j)$
6.     Remove node $q$ and use Dijkstra's algorithm to compute shortest path $m_{i,j}$ from each source i to every other node j
7.     Invert the reweigh transform $\forall i, j, m_{i,j} \leftarrow m_{i,j} + d(j) - d(i)$
8.     return $Strengthening(m, dim)$
```

A.2. Operators

A.2.1. Meet with Linear Constraint

In this subsection, we describe meet of an octagon with a linear constraint. Algorithm 51 shows the pseudo code for meet with linear constraint. We only show "$\geq$" type of constraints for simplicity. The constraints that can be modeled as octagonal are handled exactly. For other types of constraints, an approximation is required.

Lines 6 and 8 shows the case of octagonal constraints of the form $\pm x_i + [-a, b] \geq 0$ and $\pm x_i \pm x_j + [-a, b] \geq 0$. These are handles by function *meet_oct_lincons* in Algorithm 50.

Suppose we have,

$$x_i + x_j + [-a, b] \geq 0 \tag{A.1}$$

This is equivalent to,

$$-x_i - x_j \leq b \tag{A.2}$$

Therefore, new bound for $m_{2i+1,2j}$ should be $min(m_{2i+1,2j}, b)$. The other types of octagonal constraints can be handled similarly. Incremental closure is applied on the resulting octagon with respect to one of the variable v_i or v_j. The runtime is thus domainted by incremental closure.

For non octagonal constraints, we compute the maximum of expression on left hand side of constraint using interval arithmetics at lines 12-17. Let $[r_i, s_i]$ be the bound for varibale v_i. Then the bound $[r, s]$ for expression

Algorithm 50 Meet With Octagonal Linear Constraint

```
function MEET_OCT_LINCONS(m,cons,d)
    $m \leftarrow$ input matrix
    $cons \leftarrow$ Octagonal Linear Constraint
    $d \leftarrow$ number of variables in program
    switch $cons$ do
        case $x_i + [-a, b] \geq 0$
            $m_{2i,2i+1} \leftarrow min(m_{2i,2i+1}, 2b)$
            $incr\_closure(m, i, d)$
        case $-x_i + [-a, b] \geq 0$
            $m_{2i+1,2i} \leftarrow min(m_{2i+1,2i}, 2b)$
            $incr\_closure(m, i, d)$
        case $x_i + x_j + [-a, b] \geq 0$
            $m_{2i,2j+1} \leftarrow min(m_{2i,2j+1}, b)$
            $incr\_closure(m, j, d)$
        case $-x_i + x_j + [-a, b] \geq 0$
            $m_{2i+1,2j+1} \leftarrow min(m_{2i+1,2j+1}, b)$
            $incr\_closure(m, j, d)$
        case $x_i - x_j + [-a, b] \geq 0$
            $m_{2i,2j} \leftarrow min(m_{2i,2j}, b)$
            $incr\_closure(m, j, d)$
        case $-x_i - x_j + [-a, b] \geq 0$
            $m_{2i+1,2j} \leftarrow min(m_{2i+1,2j}, b)$
            $incr\_closure(m, j, d)$
```

$$[-a_1, b_1]x_1 + [-a_2, b_2]x_2 + \ldots + [-a_d, b_d]x_d + [-a, b] \tag{A.3}$$

can be computed as

$$[r, s] \leftarrow [-a_1, b_1] \otimes [r_1, s_1] \oplus [-a_2, b_2] \otimes [r_2, s_2] \oplus \ldots \oplus [-a_d, b_d] \otimes [r_d, s_d] \oplus [-a, b] \tag{A.4}$$

where $\oplus$ and $\otimes$ denote addition and multiplication in interval domain respectively. While computing upper bound *s* for the expression we do not consider the variables for which the upper bound of,

$$[-a_i, b_i] \otimes [r_i, s_i] \tag{A.5}$$

is not finite. We also keep track of number *cinf* and location c_1, c_2 of such variables. If the value of *s* is finite then we consider three cases for approximation, the rest are left unhandled.

Line 20 handles the case in which *cinf* is zero. We derive quadratic number of bounds in this case. Let us consider the case for,

$$\begin{aligned} [-a_j, b_j]x_j + [-a_k, b_k]x_k + exp \geq 0 \\ a_j, a_k \leq -1 \\ s_j, s_k! = inf \end{aligned} \tag{A.6}$$

where *exp* denotes the rest of the expression. We derive bound as follows. We know that,

$$s \geq 0 \tag{A.7}$$

Algorithm 51 Meet with Linear Constraint

```
function MEET_LINCONS(m,cons,d)
    $m \leftarrow$ input matrix
    $cons \leftarrow$ Linear Constraint
    $d \leftarrow$ number of variables in program
    switch $cons$ do
        case $c_i x_i + [-a, b] \geq 0$                                          ▷ $c_i \in [-1, 1]$
            $meet\_oct\_lin\_cons(m, cons, d)$
        case $c_i x_i + c_j x_j + [-a, b] \geq 0$                                 ▷ $c_i, c_j \in [-1, 1]$
            $meet\_oct\_lin\_cons(m, cons, d)$
        case $[-a_1, b_1]x_1 + [-a_2, b_2]x_2 + \ldots + [-a_d, b_d]x_d + [-a, b] \geq 0$
            $s \leftarrow 2b, cinf \leftarrow 0, c_1 \leftarrow 0, c_2 \leftarrow 0$
            for $j \leftarrow 0$ to $dim$ do
                $[p_j, q_j] \leftarrow int\_mul([a_j, b_j], [m_{2j,2j+1}, m_{2j+1,2j}])$
                if $(\ is\_finite(q_j))$ then
                    $s \leftarrow s + q_j$
                else
                    $cinf \leftarrow cinf + 1, c_2 \leftarrow c_1, c_1 \leftarrow j$
            if $(\ is\_finite(s))$ then
                switch $cinf$ do
                    case 0
                        for $j \leftarrow 0$ to $dim$ do
                            if $(a_j \leq -1)\ \&\ (is\_finite(m_{2j+1,2j}))$ then
                                $t \leftarrow s - m_{2j+1,2j}$
                                $u_j \leftarrow 2j + 1$
                            else if $(b_j \leq -1)\ \&\ (is\_finite(m_{2j,2j+1}))$ then
                                $t \leftarrow s - m_{2j,2j+1}$
                                $u_j \leftarrow 2j$
                            else
                                continue
                            for $k \leftarrow 0$ to $dim$ do
                                if $(a_k \leq -1)\ \&\ (is\_finite(m_{2k+1,2k}))$ then
                                    $t \leftarrow (t - m_{2k+1,2k})/2$
                                    $m_{2k,u_j} \leftarrow min(m_{2k,u_j}, t)$
                                else if $(b_k \leq -1)\ \&\ (is\_finite(m_{2k,2k+1}))$ then
                                    $t \leftarrow (t - m_{2k,2k+1})/2$
                                    $m_{2k+1,u_j} \leftarrow min(m_{2k+1,u_j}, t)$
                    case 1
                        if $(a_{c_1} \leftarrow 1)\ \&\ (b_{c_1} \leftarrow -1)$ then
                            $u_j \leftarrow 2c_1$
                        else if $(a_{c_1} \leftarrow -1)\ \&\ (b_{c_1} \leftarrow 1)$ then
                            $u_j \leftarrow 2c_1 + 1$
                        else
                            break
                        for $k \leftarrow 0$ to $dim$ do
                            if $(a_k \leq -1)\ \&\ (is\_finite(m_{2k+1,2k}))$ then
                                $t \leftarrow (s - m_{2k+1,2k})/2$
                                $m_{2k,u_j} \leftarrow min(m_{2k,u_j}, t)$
                            else if $(b_k \leq -1)\ \&\ (is\_finite(m_{2k,2k+1}))$ then
                                $t \leftarrow (s - m_{2k,2k+1})/2$
                                $m_{2k+1,u_j} \leftarrow min(m_{2k+1,u_j}, t)$
                    case 2
                        if $(a_{c_1} \leftarrow 1)\ \&\ (b_{c_1} \leftarrow -1)$ then
                            $ui \leftarrow 2c_1$
                        else if $(a_{c_1} \leftarrow -1)\ \&\ (b_{c_1} \leftarrow 1)$ then
                            $ui \leftarrow 2c_1 + 1$
                        else
                            break
                        if $(a_{c_2} \leftarrow 1)\ \&\ (b_{c_2} \leftarrow -1)$ then
                            $u_j \leftarrow 2c_2$
                        else if $(a_{c_2} \leftarrow -1)\ \&\ (b_{c_2} \leftarrow 1)$ then
                            $u_j \leftarrow 2c_2 + 1$
                        else
                            break
                        $m_{ui\oplus 1,u_j} \leftarrow min(m_{ui\oplus 1,u_j}, s/2)$
```

Since $a_j, a_k \leq -1$, therefore in computation of s, the upper bounds of $[-a_j, b_j] \otimes [r_j, s_j]$ and $[-a_k, b_k] \otimes [r_k, s_k]$ were added to s. We have,

$$\begin{aligned} & s + x_j + x_k - x_j - x_k \geq 0 \\ & - x_j - x_k \leq s - x_j - x_k \\ & - x_j - x_k \leq s - max(x_j) - max(x_k) \end{aligned} \tag{A.8}$$

Line 37 handles the case for *cinf* = 1. In this case we derive linear number of bounds for all constraints involving variable v_{c_1} provided $a_{c_1}, b_{c_1} \in [-1, 1]$. Let us consider the case for,

$$\begin{aligned} x_{c_1} + [-a_k, b_k]x_k + exp \geq 0 \\ a_k \leq -1 \\ s_k! = inf \end{aligned} \tag{A.9}$$

Since upper bound of $[-a_{c_1}, b_{c_1}] \otimes [r_{c_1}, s_{c_1}]$ was not involved in computation of *s*, we have,

$$\begin{aligned} & x_{c_1} + s \geq 0 \\ & x_{c_1} + s + x_k - x_k \geq 0 \\ & - x_{c_1} - x_k \leq s - x_k \\ & - x_{c_1} - x_k \leq s - max(x_k) \end{aligned} \tag{A.10}$$

Line 51 handles the case when *cinf* = 2. In this case we derive only one bound for inequality involving variable v_{c_1} and v_{c_2} provided $a_{c_1}, a_{c_2}, b_{c_1}, b_{c_2} \in [-1, 1]$. Let us consider the constraint,

$$x_{c_1} + x_{c_2} + exp \geq 0 \tag{A.11}$$

Since upper bounds of both $[-a_{c_1}, b_{c_1}] \otimes [r_{c_1}, s_{c_1}]$ and $[-a_{c_2}, b_{c_2}] \otimes [r_{c_2}, s_{c_2}]$ were not involved in computation of *s*, we have,

$$\begin{aligned} & x_{c_1} + x_{c_2} + s \geq 0 \\ & - x_{c_1} - x_{c_2} \leq s \end{aligned} \tag{A.12}$$

A.2.2. Meet with Non Linear Constraint

The meet with non linear constraint first checks if the octagon is empty. It then converts non linaer constraint to linear using a linearization algorithm. *Meet_Lincons* is then applied on the linear constraint. For simplicity, we assume that the linearization algorithm returns only one linear constraint per non linear constraint. Algorithm 52 shows the pseudo code for this operator. The runtime is dominated by *is_bottom* which computes the closure.

Algorithm 52 Meet with Non Linear Constraint

```
function MEET_NON_LINCONS(m,cons,d)
    $m \leftarrow$ input matrix
    $cons \leftarrow$ Non Linear Constraint
    $d \leftarrow$ number of variables in program
    if $is\_bottom(m, d)$ then
        return $m$
    $lcons \leftarrow linearize(cons)$
    return $meet\_lincons(m, lcons, d)$
```

A.2.3. Widening

A binary operator $\triangledown$ is widening for an abstract domain ordered by $\subseteq$ if the following two conditions hold:

1. $\forall X, Y, (X \triangledown Y) \supseteq X, Y$, and

2. for every chain $(X_i)_{i \in N}$, the increasing chain $(Y_i)_{i \in N}$ defined by,

$$\begin{cases} Y_0 & \stackrel{\text{def}}{=} X_0. \\ Y_{i+1} & \stackrel{\text{def}}{=} Y_i \triangledown X_{i+1}. \end{cases} \tag{A.13}$$

is stable after a finite number of iterations, i.e., $\exists n, Y_{n+1} \leftarrow Y_n$, here $X_{i+1} \leftarrow F(X_i)$ where F is the abstract transfer function for the domain. It is usually formed by combination of different domain operators depending on statements in the analyzed program.

Widening operator is used to accelerate convergence towards fixpoint at control flow join points. At such points, the octagon computed in previous iteration is combined with the octagon computed after applying transfer function in the current iteration. It compares the corresponding values in the two input matrices, if the value in second matrix is greater than first then it sets the corresponding element in output to infinity.

Algorithm 53 Widening Operator

```
function WIDENING(m,n,o,dim)
    $m, n \leftarrow$ input matrix
    $o \leftarrow$ output matrix
    $dim \leftarrow$ number of variables in program
    $oct\_closure(n, dim)$
    for $i \leftarrow 0$ to $2 * dim$ do
        for $j \leftarrow 0$ to $2 * dim$ do
            if $m_{i,j} < n_{i,j}$ then
                $o_{i,j} \leftarrow inf$
            else
                $o_{i,j} \leftarrow m_{i,j}$
```

For the convergence of the increasing chain, it requires the first argument to not be closed. Thus, it is not possible to close the matrix obtained after widening. The reason for this is that closure and widening are conflicting operations. Closure usually increases finite values in the matrix whereas widening decreases finite values. It is however possible to close the second argument. Algorithm 53 shows the pseudo code for widening operator. The runtime is dominated by the closure operation on second matrix.

A.2.4. Bottom

The bottom operator allocates the lowest ordered element in the lattice representing octagons ordered by inclusion operator. Implementation wise, a bottom octagon is represented as *NULL*.

A.2.5. Is Bottom

This operator tests if an octagon represents bottom element. Besides being *NULL*, an octagon can also be bottom if the system of constraints represented by it is inconsistent. Therefore, the octagon is first closed to see if a negative cycle is detected. Algorithm 54 shows the pseudo code for this operator. The closure makes it an expensive operator in the octagon domain.

Algorithm 54 Is Bottom Operator

```
1. function IS_BOTTOM(m,dim)
2.     $m \leftarrow$ input matrix
3.     $dim \leftarrow$ number of variables in program
4.     if $m \leftarrow NULL$ then return true
5.     $o \leftarrow oct\_closure(m, dim)$
6.     if $o == Bottom$ then return true
    return false
```

A.2.6. Saturate Linear Constraint

This operator checks if an octagon saturates a linear constraint. The operator only handles constraints having expressions of the form $[-a, b]$, $\pm x_i + [-a, b]$ and $\pm x_i \pm x_j + [-a, b]$. For other expressions *false* is returned by default. For the sake of simplicity, we assume $\geq$ constraints. The other type of constraints can be handled similarly.

Suppose we have

$$x_i + x_j + [-a, b] \geq 0 \tag{A.14}$$

.

This can be written as,

$$-x_i - x_j \leq [-a, b] \tag{A.15}$$

.

Now, if from the octagon we have the constraint,

$$-x_i - x_j \leq c \tag{A.16}$$

.

The octagon saturates the given constraint if,

$$\begin{aligned} c &\leq -a \\ c + a &\leq 0 \end{aligned} \tag{A.17}$$

Algorithm 55 shows the pseudo code for this operator. Again, the runtime of this operator is dominated by the time to compute closure.

Algorithm 55 Saturate Linear Constraint

```
function SAT_LINCONS(m,cons,dim)
    $m \leftarrow$ input matrix
    $cons \leftarrow$ Linear Constraint
    $dim \leftarrow$ number of variables in program
    $oct\_closure(m, dim)$
    if $cons \leftarrow ([-a, b] \geq 0)$ then
        return $a \leq 0$
    else if $cons \leftarrow (x_i + [-a, b] \geq 0)$ then
        return $m_{2i,2i+1} + 2a \leq 0$
    else if $cons \leftarrow (-x_i + [-a, b] \geq 0)$ then
        return $m_{2i+1,2i} + 2a \leq 0$
    else if $cons \leftarrow (x_i + x_j + [-a, b] \geq 0)$ then
        return $m_{2i,2j+1} + a \leq 0$
    else if $cons \leftarrow (-x_i + x_j + [-a, b] \geq 0)$ then
        return $m_{2i+1,2j+1} + a \leq 0$
    else if $cons \leftarrow (x_i - x_j + [-a, b] \geq 0)$ then
        return $m_{2i,2j} + a \leq 0$
    else if $cons \leftarrow (-x_i - x_j + [-a, b] \geq 0)$ then
        return $m_{2i+1,2j} + a \leq 0$
    else if $cons\ has\ more\ than\ two\ variables$ then
        return false
```

A.2.7. Saturate Non Linear Constraint

This operator checks if an octagon saturates a non-linear constraint. It first linearizes the non linear constraint using a linearization algorithm. For simplicity, we assume that the linearization algorithm returns only one linear constraint per non linear constraint. It then applies *sat_lincons* function defined in previous section on the linearized constraint. Algorithm 56 shows the pseudo code for this operator. The runtime is dominated by call to *is_bottom* function.

Algorithm 56 Saturate Non Linear Constraint

```
function SAT_NON_LINCONS(m,cons,dim)
    $m \leftarrow$ input matrix
    $cons \leftarrow$ Non Linear Constraint
    $dim \leftarrow$ number of variables in program
    if $is\_bottom(m, dim)$ then
        return true
    $lcons \leftarrow linearize(cons)$
    return $sat\_lincons(m, lcons, dim)$
```

A.2.8. Octagon to Box

The octagon to box operator converts an octagon to box (array of intervals) representation. It is used for handling statements that cannot be modeled as octagonal contraints but can be handled using interval domain. An example of such a statement is the assignment statement $x \leftarrow 2y + 3z$. For this statement, the constraints for variable *x* can be obtained by extracting the intervals for variables *y* and *z* and then using interval arithmetics on the obtained intervals. The Conversion to intervals does incur precision lost as the relational information between variables is lost. Algorithm 57 shows the pseudo code for octagon to box conversion operator.

Again, the running time of the operator is dominated by time taken for computing closure. The remaining linear part accesses the elements of the matrix diagonally and may cause TLB misses

Algorithm 57 Octagon to Box

```
1. function OCT TO BOX(m,in,dim)
2.     $m \leftarrow$ input matrix
3.     $in \leftarrow$ output interval array
4.     $dim \leftarrow$ number of variables in program
5.     $oct\_closure(m, dim)$
6.     for $i \leftarrow 0$ to $dim$ do
7.         $in[i] \leftarrow [-(m_{2i,2i+1})/2, (m_{2i+1,2i})/2]$
```

for very large matrices.

A.2.9. Octagon to Array of Linear Constraints

This operator is used to extract linear constraints from an octagon. It is mainly used for printing invariants in the form of constraints between variables at different program points. Algorithm 59 shows the pseudo code for this operator.

Algorithm 58 Create Linear Constraint

```
1. function CREATE_LINCONS(i,j,c)
2.     $i, j \leftarrow$ index of variables involved in constraint
3.     $c \leftarrow$ value of constant
4.     if $i \leftarrow (j \oplus 1)$ then
5.         if $i$ is odd then
6.             return $-x_i + (c/2) \geq 0$
7.         else
8.             return $x_i + (c/2) \geq 0$
9.     else
10.        if $i$ is odd then
11.            if $j$ is odd then
12.                return $-x_i + x_j + c \geq 0$
13.            else
14.                return $-x_i - x_j + c \geq 0$
15.        else
16.            if $j$ is odd then
17.                return $x_i + x_j + c \geq 0$
18.            else
19.                return $x_i - x_j + c \geq 0$
```

Algorithm 59 Octagon to Array of Linear Constraints

```
1. function OCT TO LINCONS ARRAY(m,cons,dim)
2.     $m \leftarrow$ input matrix
3.     $cons \leftarrow$ output array of linear constraints
4.     $dim \leftarrow$ number of variables in program
5.     $n \leftarrow 0$
6.     for $i \leftarrow 0$ to $2 * dim$ do
7.         for $j \leftarrow 0$ to $2 * dim$ do
8.             if $((i! = j)\ \&\ (m_{i,j}! = inf))$ then
9.                 $cons[n] \leftarrow create\_lincons(i/2, j/2, m_{i,j})$
10.                $n \leftarrow n + 1$
```

The running time of this operator can be reduced significantly if the the matrix is sparse and a sparse data structure like the index we used for closure is available. Since this operator is mainly used for printing purposes we do not optimize it.

A.2.10. Add Dimensions

This operator is used to add variables to an octagon. Algorithm 60 shows the pseudo code for this operator. The array *arr* contains the indices of new variables in increasing order. New rows and columns are inserted at positions specified in the index. If the index is *v* then new *2v*and *2v+1*-th row and column are inserted. The rows below the added ones are shifted down. Similarly columns to the right of newly added columns in the original matrix are shifted to the right in the new matrix. The added rows and columns contain *infinity* values except at the main diagonal where 0 is inserted.

Algorithm 60 Add Dimensions

```
function ADD_DIMENSIONS(m,o,arr,nb,dim)
    $m \leftarrow$ input matrix
    $o \leftarrow$ output matrix
    $arr \leftarrow$indices of variables
    $nb \leftarrow$ number of variables to add
    $dim \leftarrow$ number of variables in program
    $oct\_closure(m, dim)$
    $Top(o, dim + nb)$
    $k \leftarrow 0$
    $ni \leftarrow 0$
    for $i \leftarrow 0$ to $2 * dim$ do
        if $((i \geq 2 * arr[k])\ \ \&\ \ (k < nb))$ then
            while $(arr[k + 1] = arr[k])\ \ \&\ \ (k < nb - 1)$ do
                $ni \leftarrow ni + 2$
                $k \leftarrow k + 1$
            $ni \leftarrow ni + 2$
            $k \leftarrow k + 1$
        $kk \leftarrow 0$
        $nj \leftarrow 0$
        for $j \leftarrow 0$ to $2 * dim$ do
            if $((j \geq 2 * arr[kk])\ \ \&\ \ (kk < nb))$ then
                while $(arr[kk + 1] = arr[kk])\ \ \&\ \ (kk < nb - 1)$ do
                    $nj \leftarrow nj + 2$
                    $kk \leftarrow kk + 1$
                $nj \leftarrow nj + 2$
                $kk \leftarrow kk + 1$
            $o_{(i+ni,j+nj)} \leftarrow m_{i,j}$
```

The runtime is dominated by the closure operation.

A.2.11. Remove Dimensions

This operator is used to remove variables from an octagon. Algorithm 61 shows the pseudo code for this operator. The array *arr* contains the indices of variables to be removed in strictly increasing order. Variables are removed from rows and columns specified by the index. If the index is *v* then *2v*-th row and column are removed. The rows below the removed ones are shifted up. Similarly, the columns to the right of removed columns are shifted to the left.

The runtime is dominated by the closure operation.

Algorithm 61 Remove Dimensions

```
1.  function REMOVE_DIMENSIONS(m,o,arr,nb,dim)
2.      $m \leftarrow$ input matrix
3.      $o \leftarrow$ output matrix
4.      $arr \leftarrow$indices of variables
5.      $nb \leftarrow$ number of variables to remove
6.      $dim \leftarrow$ number of variables in program
7.      $oct\_closure(m, dim)$
8.      $k \leftarrow 0$
9.      $i \leftarrow 0$
10.     for $ni \leftarrow 0$ to $2 * (dim - nb)$ do
11.         while $((i \geq 2 * arr[k])\ \&\ (k < nb))$ do
12.             $i \leftarrow i + 2$
13.             $k \leftarrow k + 1$
14.         $kk \leftarrow 0$
15.         $j \leftarrow 0$
16.         for $nj \leftarrow 0$ to $2 * (dim - nb)$ do
17.             while $((j \geq 2 * arr[kk])\ \&\ (kk < nb))$ do
18.                 $j \leftarrow j + 2$
19.                 $kk \leftarrow kk + 1$
20.             $o_{ni,nj} \leftarrow m_{i,j}$
21.             $j \leftarrow j + 1$
22.         $i \leftarrow i + 1$
```

A.2.12. Permute Dimensions

This operator is used to permute the positions of variables in octagon. Algorithm 62 shows the pseudo code for this operator. The array *arr* stores the mapping from current position to new position for each variable. The element *arr[i]* in the array contains the new position of variable v_i. As a result of this operation, a constraint $x_i - x_j \leq c$ is transformed to $x_{arr[i]} - x_{arr[j]} \leq c$.

Algorithm 62 Permute Dimensions

```
1.  function PERMUTE_DIMENSIONS(m,o,arr,dim)
2.      $m \leftarrow$ input matrix
3.      $o \leftarrow$ output matrix
4.      $arr \leftarrow$map from current index to new permuted index
5.      $dim \leftarrow$ number of variables in program
6.      for $i \leftarrow 0$ to $dim$ do
7.          $ni \leftarrow 2 * arr[i]$
8.          for $j \leftarrow 0$ to $dim$ do
9.              $nj \leftarrow 2 * arr[j]$
10.             $o_{ni,nj} \leftarrow m_{2i,2j}$
11.             $o_{ni,nj+1} \leftarrow m_{2i,2j+1}$
12.             $o_{ni+1,nj} \leftarrow m_{2i+1,2j}$
13.             $o_{ni+1,nj+1} \leftarrow m_{2i+1,2j+1}$
```

A.2.13. Assignment with Non Linear Expression

The operator is used to model assignment $x := expr$ to a variable x from an expression $expr$ where $expr$ can be non linear. Algorithm 63 shows the pseudo code for this operator. Since $expr$ can also contain variable x, an extra variable x' is added to the octagon using function $add\_dimensions$ which represents instance of x on the left hand side. The assignment statement is converted into a non linear inequality. Meet with resulting non linear constraint is applied on the octagon using function $meet\_non\_lincons$. The positions of x and x' are then swapped using function $permute\_dimensions$. Since x' contains the new value of original variable x, x is removed from the octagon.

Algorithm 63 Assignment with Non Linear Expression

```
1. **function** ASSIGN_NON_LINEAR_EXPR(*m,o,expr,v,dim*)
2.     $m \leftarrow$ *input matrix*
3.     $o \leftarrow$ *output matrix*
4.     $expr \leftarrow$ *right hand side of assignment statement*
5.     $v \leftarrow$ *index of variable on left hand side of assignment statement*
6.     $dim \leftarrow$ *number of variables in program*
7.     **if** $is\_bottom(m, dim)$ **then**
8.         **return** $m$
9.     $d_0 \leftarrow v$
10.    **for** $i \leftarrow 0$ **to** $dim$ **do**
11.        $p_i \leftarrow i$
12.    $p_v \leftarrow d$
13.    $p_d \leftarrow v$
14.    $add\_dimensions(m, o, d, 1, dim)$
15.    $cons \leftarrow (expr - x_{p_v} = 0)$
16.    $meet\_non\_lincons(o, cons, dim + 1)$
17.    $permute\_dimensions(o, o, p, dim + 1)$
18.    $remove\_dimensions(o, o, d, 1, dim + 1)$
19.    **return** o
```

This operator involves a seuence of different octagon operators which in turn use octagon closure. Thus optimizing closure for performance optimizes assignment as well.