Making Numerical Program Analysis Fast

Gagandeep Singh
Department of Computer Science
ETH Zürich, Switzerland
gsingh@inf.ethz.ch

Markus Püschel
Department of Computer Science
ETH Zürich, Switzerland
pueschel@inf.ethz.ch

Martin Vechev
Department of Computer Science
ETH Zürich, Switzerland
martin.vechev@inf.ethz.ch

Abstract

Numerical abstract domains are a fundamental component in modern static program analysis and are used in a wide range of scenarios (e.g. computing array bounds, disjointness, etc). However, analysis with these domains can be very expensive, deeply affecting the scalability and practical applicability of the static analysis. Hence, it is critical to ensure that these domains are made highly efficient.

In this work, we present a complete approach for optimizing the performance of the Octagon numerical abstract domain, a domain shown to be particularly effective in practice. Our optimization approach is based on two key insights: i) the ability to perform *online decomposition* of the octagons leading to a massive reduction in operation counts, and ii) leveraging classic performance optimizations from linear algebra such as vectorization, locality of reference, scalar replacement and others, for improving the key bottlenecks of the domain. Applying these ideas, we designed new algorithms for the core Octagon operators with better asymptotic runtime than prior work and combined them with the optimization techniques to achieve high actual performance. We implemented our approach in the Octagon operators exported by the popular APRON C library, thus enabling existing static analyzers using APRON to immediately benefit from our work.

To demonstrate the performance benefits of our approach, we evaluated our framework on four published static analyzers showing massive speedups for the time spent in Octagon analysis (e.g., up to 146x) as well as significant end-to-end program analysis speedups (up to 18.7x). Based on these results, we believe that our framework can serve as a new basis for static analysis with the Octagon numerical domain.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; F.2.1 [*Numerical Algorithms and Problems*]: Computations on matrices

Keywords Fast numerical program analysis, octagon abstract domain, octagon decomposition, vectorized octagon operators, sparse octagon operators, octagon closure algorithm

PLDI'15, June 13–17, 2015, Portland, OR, USA
Copyright 2015 ACM 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2738000

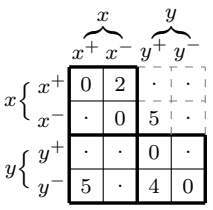
1. Introduction

Abstract interpretation is a general theory for approximating program semantics [11] and forms the basis of many static program analyzers [4, 7, 13, 21]. The fundamental concept in the theory of abstract interpretation is the *abstract domain*, a mathematical representation of program properties equipped with a set of operators. Over the last few decades, the research community has developed a wide range of abstract domains targeting a diverse set of important program properties including heap [26, 28], numerical properties [12, 20, 22, 23], termination [29, 30] and many others. In this work we focus on numerical domains, which are at the core of any modern static analyzer [6–8, 13, 24, 25].

Performance gap. One of the fundamental tensions in abstract interpretation and static analysis in general is coming up with abstract domains that are both precise enough to capture interesting properties yet are scalable and efficient enough to handle real-world programs. This trade-off is particularly stark in the case of numerical domains where over the years, researchers have devised various numerical abstract domains [20, 22, 23] that are less expressive than the powerful Polyhedra [12], yet are more efficient (and still precise enough) to handle larger and more complex programs. However, there is still a large gap between the mathematical definition of an abstract domain and how this domain is actually implemented. Addressing this gap is important and may be the difference between being able to analyze a program in reasonable time or not at all.

This work. In this work, we present a comprehensive end-to-end approach for optimizing the performance of the Octagon abstract domain, a widely used powerful relational domain which strikes a balance between expressivity and efficiency of its operators. Our work is based on two key insights. First, we observe that we can *decompose* octagons *during* program analysis *and* incrementally maintain and update this decomposition as the analysis proceeds, without incurring prohibitive overhead. This enables us to design octagon data structures and operators which leverage this decomposition and work on "smaller decomposed pieces" rather than on the entire monolithic octagon. Second, we exploit the matrix-based representation of octagons to apply known numerical code optimizations from high performance linear algebra [16] such as vectorization, locality of reference, scalar replacement and others. As we will show, our work leads to significant benefits both in asymptotic and actual runtime when analyzing real-world programs.

We implemented our new octagon operators as part of the APRON C library, preserving its existing API. This means that our new library can be directly plugged into any program analyzer already using APRON, which we demonstrate by running several existing analyzers [6, 8, 24, 25], showing massive speedups for the Octagon operators as well as significant speedups for end-to-end analysis times.



Encoded Octagon Inequalities

$-2x \leq 2$	$y - x \leq \infty$	$-y - x \leq \infty$
$2x \leq \infty$	$y + x \leq 5$	$x - y \leq \infty$
$x - y \leq \infty$	$-x - y \leq \infty$	$-2y \leq \infty$
$x + y \leq 5$	$y - x \leq \infty$	$2y \leq 4$

Figure 1: An example DBM encoding octagonal inequalities between variables x and y.

Main contributions:. In summary we contribute:

- New data structures and algorithms for octagons and the associated operators of the Octagon domain. Our algorithms perform and maintain octagon decomposition online during analysis, and incrementally maintain that decomposition as the analysis proceeds. This enables operators to focus only on the relevant parts of the octagon and to perform even asymptotically less work than state-of-the-art operators (which operate on the entire octagon).
- A new algorithm for computing octagon closure (the most expensive operator) which, besides the above reductions, reduces the cost to half. Further, its implementation uses state-of-the art optimizations from linear algebra performance libraries.
- A complete end-to-end implementation of our algorithms and optimizations inside the APRON library, enabling existing analyzers based on APRON to directly use our implementation without any changes.
- A thorough experimental evaluation showing massive speedups of Octagon operators on realistic programs as well as significant end-to-end speedups for several state-of-the-art static analyzers.

2. Background: Octagon Abstract Domain

In this section we provide the necessary background on the Octagon abstract domain [23]. We provide basic definitions and introduce the most important operators using a simple example. Finally, we identify key optimization opportunities, which are then explored in the remainder of the paper.

2.1 Octagon Abstract Domain

The Octagon domain is a relational numerical abstract domain supporting binary inequalities of the form $a_i v_i - a_j v_j \leq c$. Here, v_i and v_j are variables, $a_i, a_j \in \{-1, 0, 1\}$ are coefficients, and $c \in \mathbb{R} \cup \{\infty\}$ is the bound of the inequality. Note that the inequality with $c = \infty$ always holds. An element in the octagon domain is a conjunction of such inequalities and called octagon. An example is the formula $(x + y \leq 5) \wedge (y + z \leq 3)$, while the formula $(5x + y \leq 7)$ is not because the coefficient 5 is disallowed by the Octagon domain. In two dimensions (i.e., two variables), such inequalities can describe (the usual) octagons, which motivates the name.

Representing octagons. The data structures commonly used to encode octagons are difference bound matrices (DBMs) [23]. Let $V = \{v_0, \ldots, v_{n-1}\}$ denote a set of n variables. For each variable v_i, we introduce two variables v_i^+ and v_i^-, where $v_i^+ = v_i$ and $v_i^- = -v_i$. We obtain the extended variable set $\hat{V} = \{\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_{2n-1}\}$ where $\hat{v}_{2i} = v_i^+$ and $\hat{v}_{2i+1} = v_i^-$, $0 \leq i < n$.

A full DBM is a $2n \times 2n$-matrix O. The matrix entry $O_{i,j} = c$ encodes the inequality $\hat{v}_j - \hat{v}_i \leq c$, $0 \leq i, j < 2n$. Fig. 1 shows an example DBM and the associated encoded inequalities. For simplicity we use the symbol $\cdot$ instead of ∞. For example, $O_{1,2}$ represents the inequality $y^+ - x^- \leq 5$, i.e., $y + x \leq 5$. Since $0 \leq 0$, all $O_{i,i} = 0$. Note that the inequality $-2x \leq 2$ arises from $x^- - x^+ \leq 2$.

Redundancy in representation. The full DBM encodes $4n^2$ inequalities but contains redundancy: at most $2n^2 + 2n$ are unique. Namely, $v_j - v_i \leq c$ can be represented either as $v_j^+ - v_i^+ \leq c$ or as $v_i^- - v_j^- \leq c$. For example, in Fig. 1, $x + y \leq 5$ is represented by both entries $O_{1,2}$ and $O_{3,0}$: $O_{3,0}$ encodes $x^+ - y^- \leq 5$ and $O_{1,2}$ encodes $y^+ - x^- \leq 5$. In general, $O_{i,j}$ and $O_{j \oplus 1, i \oplus 1}$ (where $\oplus$ means bitwise xor) encode the same inequality. Thus, if we view the DBM as a matrix of 2×2 submatrices, this redundancy can be exploited by storing and maintaining only the lower triangular part, marked with bold lines in Fig. 1.

In this work we distinguish between *trivial* and *non-trivial* Octagon inequalities. Trivial inequalities are always true (e.g., $v_i - v_j \leq \infty$ or $0 \leq 0$) and impose no constraint on the variables. A non-trivial inequality, however, does not always hold (e.g. $v_i - v_j \leq 2$) and hence constrains the set of values that the variables can take. The top (maximal) element $\top$ in the Octagon domain imposes no constraints ($O_{i,i} = 0$, $O_{i,j} = \infty$) and thus contains all other octagons.

Octagon operators. The Octagon abstract domain (like any abstract domain) is equipped with a number of operators that are used for building program analyzers. These operators capture the effect of standard programming constructs (e.g., assignments, conditionals) in the abstract domain. Practical libraries implementing Octagons such as APRON [19] contain many operators of various complexity to support a wide range of programs.

While our implementation handles these operators, in the presentation of this work we focus on explaining the key concepts on only a representative subset consisting of Join ($\sqcup$), Widening ($\triangledown$), Meet ($\sqcap$), Top, and Closure ($(\cdot)^*$). As explained later, Closure is critical as it is the most expensive operator. Next, we informally introduce these operators using a simple example: the program shown in Fig. 2.

2.2 Program Analysis with Octagons: An Example

In a program analysis setting, usually a single octagon is maintained at each program point: it represents knowledge about the variable ranges before its execution. We refer to the octagons in our example Fig. 2 as $O_1, O_2, \ldots, O_6$. The analysis proceeds iteratively by selecting an octagon at a given program point, say O_1, applying the effect of the given program statement at that point on that octagon (i.e., `x=1`), and producing a new octagon, in this case O_2. The analysis terminates when it reaches a fixed point, i.e., when all octagons remain unchanged.

Example iteration. Upon startup, the analysis initializes all octagons to the top element: $O_i = \top$. An example is O_1 in Fig. 2. Next, the analysis processes the statement `x=1`. This assignment imposes two new inequalities $2x \leq 2$ and $-2x \leq -2$ on the octagon O_1. This effect is captured by computing the intersection of these two inequalities with O_1. This intersection is computed using the Meet ($\sqcap$) operator. For two same inequalities on the same variables, meet outputs the one with the smaller bound. For example, the meet of $2x \leq \infty$ represented in O_1 with the inequality $2x \leq 2$ results in the inequality $2x \leq 2$ stored in the resulting matrix O_2. In this case, meet requires constant time; in general, the meet of two octagons has an asymptotic complexity of $O(n^2)$.

For the next assignment statement `y=x`, the constraints $x - y \leq 0$ and $y - x \leq 0$ induced by that statement are similarly intersected with the octagon O_2 again via the meet operator, resulting in the octagon O_3. Note that since the analysis has not yet encountered the variable m, the DBM O_3 does not contain non-trivial inequalities involving that variable.

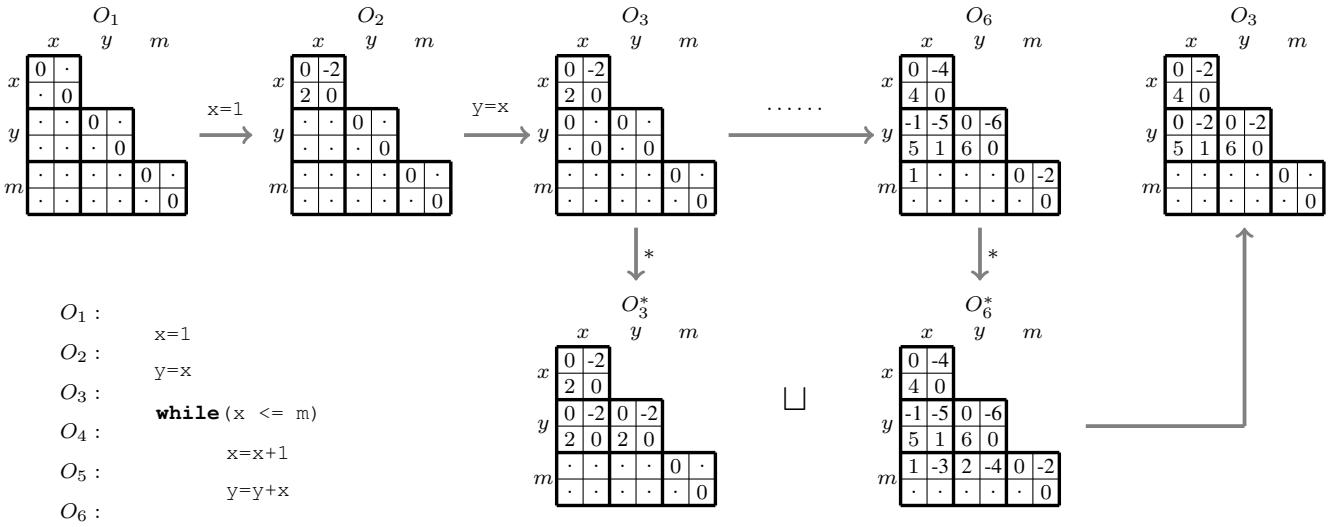


Figure 2: Octagon analysis (first iteration) on the example program (bottom left).

Algorithm 1 Octagon Closure on full DBM

```
1: function OCTAGON_CLOSURE_FULL(O,n)
2:     Parameters:
3:         O ← input full DBM
4:         n ← number of variables
5:     for k ∈ {0, 1, ... 2n − 1} do          ▷ Shortest path closure
6:         for i ∈ {0, 1, ... 2n − 1} do
7:             for j ∈ {0, 1, ... 2n − 1} do
8:                 O_{i,j} = min(O_{i,j}, O_{i,k} + O_{k,j})
9:     for i ∈ {0, 1, ... 2n − 1} do          ▷ Strenghtening step
10:        for j ∈ {0, 1, ... 2n − 1} do
11:            O_{i,j} = min(O_{i,j}, (O_{i,i⊕1} + O_{j⊕1,j})/2)
```

The analysis proceeds with interpreting the statements inside the loop body. We omit the resulting matrices O_4 and O_5 and only show the final DBM O_6. The analysis now returns to the loop head and propagates the octagon O_6 to that point. To compute the new result at the loop head, the analysis now has to compute the union of the matrix O_6 with the previous matrix at that point, O_3. The union of two octagons is in general not an octagon; thus an overapproximation is computed via the Join ($\sqcup$) operator. The join operator is conceptually simple: it produces a new matrix where each entry in that matrix is the corresponding maximum entry of the two input matrices.

Closure. The degree of over-approximation while computing the join can be reduced by first applying the so called closure operator on both O_3 and O_6, and then joining the resulting octagons. In fact, several Octagon operators besides join use closure to obtain better precision. Informally, the closure operator adds all constraints to a DBM that can be derived by others. Its pseudo-code on full DBMs is shown in Algorithm 1 (due to [3]). It consists of a shortest path closure step which minimizes inequalities transitively and a strengthening step. Importantly, the closure has *cubic complexity* in n. It is the only operator with this property and thus the single most expensive one for the Octagon domain.

Returning to our example, for O_3 the shortest path step of the closure combines $y - x \leq 0$ and $x \leq 1$ to obtain the inequality $y \leq 1$. The strengthening step combines $x \leq 1$ and $y \leq 1$ in O_3 to obtain $x + y \leq 2$. The closed O_3 and O_6, shown at the bottom in Fig. 2 as O_3^* and O_6^*, are then joined to produce the new octagon O_3 (rightmost in Fig. 2). The join operator takes the maximum of each entry, analogous to the meet operator, and has worst-case quadratic complexity in n.

Widening. As usual in program analysis, one needs to make sure that the analysis terminates in a timely manner. In particular for loops, this is done with the widening operator that accelerates convergence. Specifically, if the bound of an inequality keeps increasing in every iteration then the widening operator sets it after a few steps to ∞. This operator has an asymptotic complexity of $O(n^2)$.

2.3 Performance Optimization: Opportunities and Challenges

The goal of this paper is to considerably improve the performance of program analyzers that use the Octagon domain. We do this by identifying and exploiting redundancy and structure when performing the Octagon operators, combined with classic architecture-cognizant optimization techniques commonly used in high-performance numerical libraries. We identify the following main opportunities:

- During program analysis, many variable pairs are *not related* by inequalities, i.e., the DBM is sparse in the sense that many entries are $= \infty$. Exploiting this sparseness can significantly reduce the cost of operators.
- A more specific case of sparseness occurs when the set of variables decomposes into subsets that are mutually unrelated to each other. In this case, the expensive closure can be decomposed to work on these smaller components.
- The matrix-based representation of octagons and operators should make common numerical performance optimizations like vectorization, locality of reference, scalar replacement, applicable. In particular, these can be carefully designed and applied to optimize the closure operator, which is the key bottleneck.

Exploiting these opportunities poses significant challenges. In contrast to other domains of numerical computing, the above mentioned matrix structures continually change *dynamically* during program analysis. Thus, these changes have to be performed ef-

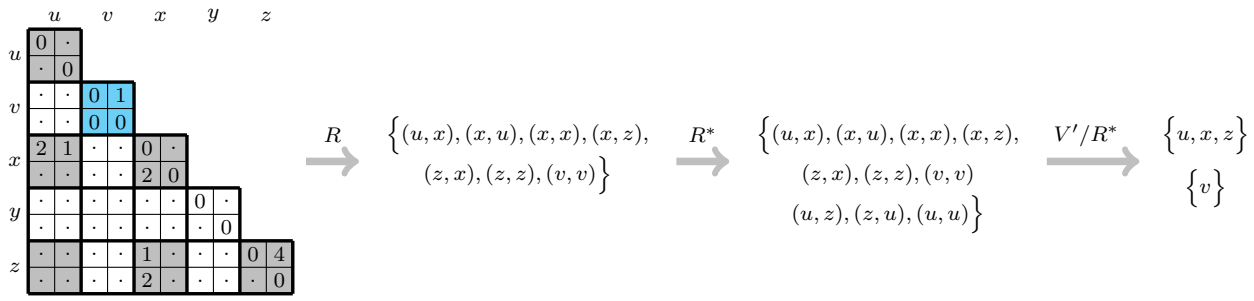


Figure 3: Computation of independent components for an octagon matrix.

ficiently, in turn requiring appropriate data structures. Accordingly, all supported Octagon operators have to be modified or redesigned to exploit the underlying structure and to maintain that structural information for subsequent operators performed in the analysis.

3. Leveraging Octagon Structure

In this section we present new data structures which leverage the structure of octagons. Our solution accounts for the different DBM structures that occur during analysis by introducing four DBM types: *Dense*, *Sparse*, *Decomposed* and *Top*. The key ideas are:

- To enable efficient switching between data types *during* analysis, each type includes a complete DBM (with all n variables) augmented with additional information depending on the structure. In particular, the number *nni* of entries $< \infty$ is maintained in all types.
- The DBM in each type is pre-allocated but initialized incrementally on-demand. Specifically, trivial entries are inserted only if needed.
- The additional information maintained by each type is used to design more efficient octagon operators (explained in next section).

Next, we present the different DBM data types and explain how we switch between types.

3.1 Dense

In the absence of structure, octagons are represented as dense DBMs (as in Fig. 1 and Fig. 2). This is the encoding of octagons commonly used in practice, including in APRON. We augment the DBM with the number of entries $< \infty$, called *nni* (number of non-infinity values).

3.2 Sparse

We introduce a special type to represent sparse DBMs. There are various sparse representations, widely used in linear algebra, including compressed sparse row (CSR) and adjacency lists. However, the former does not support an efficient change of the sparsity pattern, and both do not support an efficient conversion to a dense matrix which is important to address as this change may occur during the program analysis. Thus, as mentioned already, we use a complete DBM (as in Fig. 2) augmented with the number of elements that are $< \infty$. We do not maintain an index of the non-trivial entries as this would lead to a quadratic space overhead. But, as explained later, we do compute such an index in conjunction with each closure to reduce computation cost.

3.3 Decomposed

In octagons that occur during the analysis of real-world programs, the set of variables tends to decompose into groups where variables in a group are related via inequalities, while variables from different groups are not related via any inequality. We refer to such a group as an *independent component*. These independent components can be used to decompose a large octagon into smaller octagons, and in particular the closure can be decomposed accordingly, resulting in significant performance improvements.

Decomposition vs. variable packing. The concept of decomposing a large octagon into smaller octagons has been applied previously using variable packing [32]. However, in the case of variable packing, the decomposition is done *before* running the analysis and is thus not based on abstract semantics. Importantly, two variables in separate packs could become connected during analysis even though they are not related in the program. As a result, the analysis may *lose overall precision*.

In contrast, our approach *dynamically* decomposes octagons during analysis according to the semantics of octagon operators and *guarantees* that if two variables are not in the same independent component they will only have trivial inequalities. Thus, the overall precision of the analysis is not affected.

Formal definition of independent components. Let O be the given octagon and V be the set of variables at a given program point in the analysis. The set of non-trivial inequalities C_{v_1,v_2} between two variables v_1 and v_2 is:

$$C_{v_1,v_2} = \begin{cases} \{\pm v_1 \pm v_2 \leq c \in O \mid c \neq \infty\}, & v_1 \neq v_2, \\ \{\pm 2v \leq c \in O \mid c \neq \infty\}, & v = v_1 = v_2. \end{cases}$$

We define a connectivity relation R on variables:

$$R = \{(v_1, v_2) \mid v_1, v_2 \in V,\ C_{v_1,v_2} \neq \emptyset\}$$

We compute the reflexive, transitive closure R^* of R which is an equivalence relation on possibly a subset $V' \subseteq V$. Then, the independent components are the elements of the induced partition V'/R^*.

Example. Consider the DBM shown on the left in Fig. 3 for $V = \{u, v, x, y, z\}$. Here, u, x and x, z participate in non-trivial inequalities among themselves as shown in R. In R^*, by transitivity, u and z will be connected as well. Note how the variable y does not participate in R, that is, $V' = V \setminus \{y\}$. As a result, we obtain two independent components $\{u, x, z\}$ and $\{v\}$; the associated smaller DBMs are colored as gray and blue in the DBM.

Data Structure. The *Decomposed* type stores the DBM, possibly not fully initialized, while the independent components are stored as a linked list of linked lists of variable indices (here, the linked lists of variable indices are sorted). In addition, the number of matrix entries $< \infty$ are also stored.

Because independent components correspond to submatrices in the complete DBM, these submatrices themselves can be either *Dense* or *Sparse*. Since this information is only relevant for the expensive (cubic complexity) closure, the exact sparsity is com-

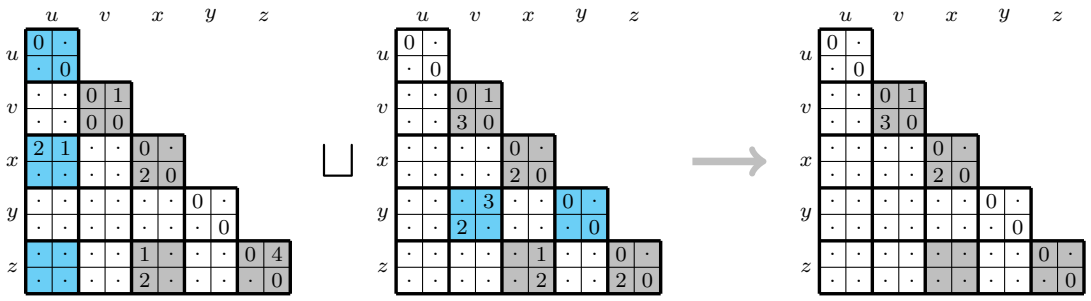


Figure 4: Join of two octagons using independent components. Gray entries in matrices correspond to the intersection of independent components. Blue entries in input matrices correspond to independent components that are not in the intersection.

puted on-the-fly prior to each closure. We explain how this is accomplished in later sections.

3.4 Top

The Top type is used to represent the highest (least precise) element in the Octagon domain. It includes a matrix that is allocated uninitialized and contains an empty set of independent components (e.g., O_1 in Fig. 2).

3.5 Switching between DBM Types

When performing program analysis with the Octagon domain, all DBMs are usually initialized to the *Top* type. As the analysis proceeds, DBMs tend to become more dense. A typical progression of types would be starting from *Top* to *Decomposed* to *Dense*. Because each type keeps the DBM, switching between types is as simple as discarding the additional information, possibly accompanied with initialization of entries to ∞. Such initialization could happen for example, when converting from a *Decomposed* to a *Dense* type (a *Dense* type requires a fully initialized matrix).

To decide when to switch from *Sparse* or *Decomposed* to a *Dense* type and vice versa, we consult the value *nni* of entries $< \infty$. Specifically, we define the sparsity of a DBM as:

$$D = 1 - (nni\ /(2n^2 + 2n))$$

and use *Dense* type if $D < t, 0 \leq t \leq 1$ (e.g., $t = 3/4$).

However, during program analysis, the sparsity can also increase again, i.e., entries in the matrix that are $< \infty$ can again become ∞. This can happen for instance when the widening operator is applied. Recovering this sparsity information exactly, and the potential independent components it might produce, has a worst case complexity of $O(n^2)$.

To avoid this cost, we only perform the exact re-computation of sparsity and independent components (only if needed) by piggybacking on the (expensive) closure operator. As closure is a common subroutine for many operators, it follows that the sparsity information of the DBM will be always up-to-date or close to it. For similar reasons, we also choose closure computations as switching points.

4. Octagon Operators

Next, for each data type, we describe the corresponding algorithms of the octagon operators. The key ideas are:

- Operators for the *Dense* type are vectorized.
- The closure operator can be specialized for the *Sparse* type to reduce operation count.
- Operators can be applied independently on submatrices of the *Decomposed* type, reducing operation count.

4.1 Operators for Dense Type

The operators for the *Dense* type are vectorized using Intel's AVX intrinsics. We present dense closure in Section 5.2 which reduces the operation count of the current state-of-the art algorithms to half. The operators working on the *Dense* type set the value of *nni* to $2n^2 + 2n$ which is an over-approximation of the actual number of entries $< \infty$. This is a reasonable assumption as the actual number of entries $< \infty$ for the *Dense* type should be close to $2n^2 + 2n$. This also avoids the overhead of checking if an entry is $< \infty$.

4.2 Operators for Sparse Type

We do not maintain auxiliary information for tracking locations of ∞ when representing the *Sparse* type. Instead, when performing closure, we build an index for the $< \infty$ entries arising in the *Sparse* type, with quadratic time and linear space overhead. This index keeps track of locations of operands $< \infty$ for a given iteration (Section 5.3). The index is not useful for other operators and is discarded after closure. Thus, we designed a special algorithm for the closure when working with the *Sparse* type. The sparse closure already checks if an entry is $< \infty$ and thus *nni* can be calculated precisely without incurring large overheads.

4.3 Operators for Decomposed Type

Recomputing the independent components for each operator from scratch is too expensive and should be avoided. We do this by computing the independent components *incrementally*, possibly making conservative decisions, i.e, producing coarser partitions. The key benefit is that many octagon operators can then work on the submatrices represented by the independent components, significantly reducing operation count.

Cost of updates. The cost of updating the set of independent components depends on the size of the set. The number of independent components is usually small (i.e. < 5) and thus the update cost is negligible compared to the cost of octagon operators. For example, a Meet ($\sqcap$), Join ($\sqcup$) or a Widening ($\triangledown$) of two octagons O_1 and O_2 induces respectively the operators $\cup$, $\cap$ and $\cap$ on the sets of independent components. We discuss how the independent components are updated when performing closure in Section 5.4.

Octagon operators on the decomposed type. Given a set of independent components, we now discuss how to adapt the octagon operators to work with the *Decomposed* type. As noted earlier, the submatrices represented by the components can be either *Dense* or *Sparse*. As the submatrices need not be contiguous, we cannot vectorize the operators for the dense submatrices directly. A workaround is to copy the submatrix, apply the operator and then copy back the result. All octagon operators except closure are at most quadratic, thus quadratic overhead due to copying will cancel out any performance gained due to vectorization. Thus, for dense submatrices, we only vectorize the closure operator (as it is cubic).

Table 1: Cost of closure(left), $\sqcup$ (middle) and $\sqcap$ (right) operator for different matrix types.

Input Type	Closure	
	Output Type	Complexity
Top	Top	$O(1)$
Sparse	Sparse	$O(n^2 + \sum_{i=1}^{n} k_i l_i)$
Dense	Dense	$O(n^3)$
Decomposed	Decomposed	$\sum_{i=1}^{m} s_i$

Input1 Type	Input2 Type	Join ($\sqcup$)		Meet ($\sqcap$)	
		Output Type	Complexity	Output Type	Complexity
Dense	Dense	Dense	$O(n^2)$	Dense	$O(n^2)$
Dense	Top	Top	$O(1)$	Dense	$O(n^2)$
Dense	Decomposed	Decomposed	$\sum_{i=1}^{m} s_i$	Dense	$O(n^2)$
Top	Top	Top	$O(1)$	Top	O(1)
Top	Decomposed	Top	$O(1)$	Decomposed	$\sum_{i=1}^{m} s_i$
Decomposed	Decomposed	Decomposed	$\sum_{i=1}^{m} s_i$	Decomposed	$\sum_{i=1}^{m} s_i$

If the submatrix is *Sparse*, we directly use the sparse closure mentioned earlier. In summary, we apply the standard non-vectorized and non-sparse operators on submatrices except when we apply the closure operator. Finally, the number *nni* of entries $< \infty$ for the *Decomposed* type is computed as the sum of its submatrices's *nni* .

Reduction in operation count. Decomposition is key to reducing the work performed by octagon operators. Fig. 4 illustrates this for the join operator. Here, the independent components of the two input matrices are colored. Without independent components, we need to operate on the entire matrices. Since the join of two octagons is reflected as an intersection of the corresponding independent components of each matrix, the join operator need only operate on the submatrix corresponding to the intersection (shown in gray in the result of the join in Fig. 4). For the example, we need to access only 16 out of the 60 entries in both matrices.

Over-approximation. It is possible that updating the set of independent components online leads to an over-approximation of the actual set of independent components in the DBM. For example, in Fig. 4, the set of independent components computed by $\cap$ for the output matrix is $\{x, z\}$ and $\{v\}$. However, extracting independent components from the output directly produces $\{x\}$ and $\{v\}$. Thus, in this case, we computed an over-approximation of the optimal set. However, it is important note that this over-approximation *does not* affect the precision of the analysis: it only produces more operations than strictly were necessary. The exact recomputation of the components with closure makes sure that the over approximation does not quickly deteriorate to the dense case.

4.4 Operators for Top Type

The *Top* type can be seen as a degenerate case of the *Decomposed* type with an empty set of independent components. Thus, we can reuse the operators of the *Decomposed* type.

4.5 Complexity of Octagon Operators

Let s_i be the cost of applying an octagon operator on the i-th submatrix of *Decomposed* type, n be the number of variables, k_i be the total number of entries $< \infty$ in $2i$ and $(2i+1)$-th rows and l_i be the total number of entries $< \infty$ in $2i$ and $(2i+1)$-th column of a closed matrix. Table 1 shows the asymptotic complexity of the closure, join and meet operators for the different types.

We note that the cost of these operators realized in popular libraries is $O(n^3)$, $O(n^2)$ and $O(n^2)$ respectively. If a matrix of *Sparse* type contains very few entries $< \infty$, the cost of the octagon closure becomes quadratic. For a *Decomposed* type, the individual submatrices are smaller, hence even if all these submatrices are dense, we reduce operation count. We reduce operation count for $\sqcup$ except when both input matrices are of type *Dense*. For $\sqcap$, the operation count is reduced only when neither of the matrix is of a *Dense* type.

5. Optimizing Closure

In this section we present our optimizations for the closure operator. We first discuss the closure as implemented in current libraries (e.g., APRON) and then present our specialized closures for each of the four octagon types. Finally, we briefly discuss incremental closure.

5.1 Standard Octagon Closure

The standard closure was already shown in Algorithm 1, consisting of two steps. The shortest path step is the same as the classic Floyd-Warshall all-pairs shortest path [15] on a full DBM (of size $2n \times 2n$). To improve space efficiency, the octagon analysis in APRON stores only the lower triangle (half representation) of the full DBM. The full DBM representing an octagon is not symmetric which means the shortest path closure on a half representation is *not the same* as applying Floyd-Warshall directly on the lower triangle of a full DBM.

Shortest path step on half representation. The full DBM has two entries (in upper and lower triangle) encoding the same inequality. These entries may get the same update (minimization using same inequalities) in alternate iterations due to asymmetry of the full DBM. For Floyd Warshall, entries in the k-th row and column do not change during the k-th iteration, therefore, as shown in Fig. 5, $O_{2k,i\oplus 1}$ (black in lower triangle) does not change during the $(2k)$-th iteration whereas $O_{i,2k+1}$ (in upper triangle) is updated. Thus, when $O_{i,2k+1}$ is required for updating $O_{i,j}$ during the $(2k+1)$-th iteration on the half representation, the accessed entry $O_{2k,i\oplus 1}$ (in the lower triangle) is not yet in a correct state. To account for this case, as shown in Algorithm 2, APRON performs two $\min$ operations per iteration of the outermost loop which runs from 0 to $2n-1$ (thus adding almost same number of operations as Floyd-Warshall on full DBM).

Strengthening on half representation. The diagonal operands $O_{i,i\oplus 1}$ and $O_{j\oplus 1,j}$ for the strengthening step of Algorithm 1 are present in the lower triangle. Thus, strengthening can be applied on the half representation with operation count reduced to half. Overall, the standard octagon closure performs $16n^3 + 22n^2 + 6n$ operations.

5.2 Dense Closure

We now show how to reduce the operation count of shortest path closure on the *Dense* type (with half representation) to $8n^3 + 10n^2 + 2n$. We also leverage numerical performance optimizations

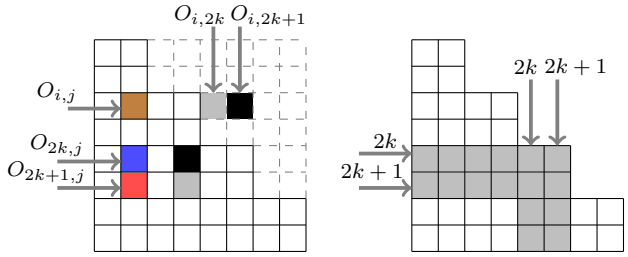


Figure 5: On left we show computation of $O_{i,j}$ for $(2k+1)$-th iteration of outermost loop of Algorithm 2. The gray matrix entries on the right are computed first for k-th iteration of the outermost loop in Algorithm 3.

Algorithm 2 APRON's Shortest Path Closure

```
1: function SHORTEST_PATH_CLOSURE_APRON(O,n)
2:     Parameters:
3:         O ← input half DBM
4:         n ← number of variables
5:     for k ∈ {0, 1, ... 2n − 1} do
6:         for i ∈ {0, 1, ... 2n − 1} do
7:             for j ∈ {0, 1, ... (i|1)} do
8:                 O_{i,j} = min(O_{i,j}, O_{i,k} + O_{k,j})
9:                 O_{i,j} = min(O_{i,j}, O_{i,k⊕1} + O_{k⊕1,j})
```

Algorithm 3 Shortest Path Closure on Dense Type

```
1: function SHORTEST_PATH_CLOSURE_DENSE(O,n)
2:     Parameters:
3:         O ← input half DBM
4:         n ← number of variables
5:     for k ∈ {0, 1, ... n − 1} do
6:         update 2k and (2k + 1) − th column
7:         update 2k and (2k + 1) − th row
8:         for i ∈ {0, 1, ... 2k − 1, 2k + 2, ... 2n − 1} do
9:             for j ∈ {0, 1, ... 2k − 1, 2k + 2, ... (i|1)} do
10:                O_{i,j} = min(O_{i,j}, O_{i,2k} + O_{2k,j})
11:                O_{i,j} = min(O_{i,j}, O_{i,2k+1} + O_{2k+1,j})
```

such as vectorization, locality of reference, scalar replacement (previously applied to Floyd Warshall on full DBMs [18]).

Reduction in operation count. The reduction is accomplished by changing the computation order of loop iterations. That is, the $2k$-th and $(2k+1)$-th iterations of the outermost loop in Algorithm 2 are performed together as a *single* iteration k in the new Algorithm 3. We first update the entries in $2k$ and $(2k+1)$-th row and column of the lower triangle as shown in Fig. 5 (right). This update requires accessing operands only from the lower triangle (thus no asymmetry issues) and can be done with one $\min$ operation per entry. Once we have updated the operands, the rest of the entries can be updated correctly. Thus, we perform two $\min$ operations per iteration of the outermost loop which now runs from 0 to $n-1$, reducing operation count by half.

Performance optimizations for shortest path closure. During the k-th iteration of the outermost loop of Algorithm 3, some of the row operands $O_{2k,j}$ and $O_{2k+1,j}$ (that may be in the upper triangle) are accessed as column entries $O_{j\oplus 1,2k+1}$ and $O_{j\oplus 1,2k}$ in the lower triangle respectively. Thus, the inner j-th loop accesses such entries columnwise. This can create a large number of cache and TLB misses which can affect performance negatively. To improve spatial locality, we perform a memory optimization where we store the updated values of $2k$ and $(2k+1)$-th column in an array before updating the remaining entries. This way the inner j-th loop accesses the entries sequentially.

Performance optimizations for strengthening. During the strengthening step, operands $O_{i,i\oplus 1}$ and $O_{j\oplus 1,j}$ are accessed diagonally which can result in cache and TLB misses. We note that the value of these diagonal operands *does not* change during strengthening. We therefore store these entries in an array before strengthening is applied. As a result, all operands are accessed sequentially during strengthening.

The performance optimizations for both steps require linear space overhead for storing the array.

Vectorization for shortest path closure. Once we have updated the entries in the $2k$ and $(2k+1)$-th row and column for the k-th iteration of the outermost loop in Algorithm 3, the remaining entries can be updated in any order. This enables a vectorized update of the remaining entries.

Vectorization for strengthening. Because the operands for the strengthening step do not change, the computation can be performed in any order. Storing all of the diagonal operands in an array allows us to access operands in contiguous order, in turn enabling vectorization of this step. Note that storing diagonal operands in an array not only improves memory performance, but also allows us to perform vectorization which would not be possible otherwise.

5.3 Sparse Closure

So far we showed how to reduce the operation count of octagon closure for dense DBMs to half. In this section, we now show how to further reduce the operation count of Algorithm 3 for *Sparse* types.

Shortest path closure. By definition, sparse matrices have a large number of ∞ values. As shortest path closure is a transitive minimization step, we need to update an entry $O_{i,j}$ only when both operands $O_{i,k}$ and $O_{k,j}$ are $< \infty$. When updating $2k$ and $(2k+1)$-th row and column during the k-th iteration of Algorithm 3, we compute an index for the location of entries $< \infty$ in the $2k$ and $(2k+1)$-th row and column. A $\min$ operation for $O_{i,j}$ is then performed only when both operands (for example $O_{i,2k}$ and $O_{2k,j}$) are present in the index.

Strengthening. The strengthening step is also a minimization step. An entry $O_{i,j}$ needs to be updated only when both operands $O_{i,i\oplus 1}$ and $O_{j\oplus 1,j}$ are $< \infty$. We maintain an index for locations of operands $< \infty$. A min operation for $O_{i,j}$ is performed only when locations of both operands $O_{i,i\oplus 1}$ and $O_{j\oplus 1,j}$ are present in the index.

Storing indices for both the shortest path closure and the strengthening step requires linear space overhead.

Complexity. The time complexity of sparse closure depends on the distribution of entries $< \infty$. For each iteration of the outermost loop of the shortest path, we have to scan two rows and columns in order to build the index. This takes linear time for each iteration and thus the complexity of sparse closure is at least quadratic (in comparison, the worst-case time complexity for dense matrices is cubic).

Performance optimizations. We perform similar performance optimizations for sparse closure as we did for the dense closure.

5.4 Decomposed Closure

Recall that the submatrices corresponding to a *Decomposed* type can be dense or sparse. Therefore, we designed the closure algorithm for the *Decomposed* type in a way which takes advantage of the sparse and dense closure algorithms described above.

Shortest path closure. This step creates a new non-trivial inequality between two variables only if there is a variable which

has a non-trivial inequality with both of them. Thus two variables in different components cannot get connected during this step (according to relation R discussed in Section 3). Hence, we can apply shortest path closure on each submatrix *independently*. For each submatrix, we first calculate the sparsity to check if it is sparse or dense. If it is sparse, we use the sparse shortest path closure. As discussed in Section 4.3, we cannot apply the vectorized dense shortest path closure on dense submatrices as they may not be contiguous. Hence we first copy the submatrix into a temporary matrix, apply vectorized Algorithm 3 on that matrix, and copy the result back to the original submatrix.

Strengthening. The strengthening step can result in merging of independent components. If there are finite diagonal operands of the form $O_{i,i\oplus 1}$ with variables belonging to different independent components, then these components are merged (via $\cup$). This results in original submatrices replaced by a new larger submatrix which may need to be partially initialized. The new submatrix is likely to be sparse and thus we use the sparse strengthening algorithm on the new submatrix.

5.5 Top Closure

A matrix of *Top* type is already closed as it has only trivial inequalities which cannot be minimized, hence we do not perform closure on octagons of this type.

5.6 Incremental Closure

It is possible to perform closure for "almost-closed" octagons in quadratic time, referred to as incremental closure. Such "almost-closed" octagons are ones for which inequalities involving very few variables are not closed. Incremental closure is usually applied after applying Meet ($\sqcap$) in the assignment statement. Incremental closure consists of a double loop which is the same as one iteration of the outermost loop of shortest path closure and a strengthening step which is the same as for the octagon closure. Thus, we can apply the same optimizations as we did for closure on all octagon types. Incremental closure is only useful when we have "almost-closed" matrices, otherwise, full closure (as discussed so far) has to be applied.

6. Experiments

In this section we evaluate the performance of our octagon operators against APRON, a widely used state-of-the-art library implementing octagons.

6.1 Experimental setup

We implemented our approach in the APRON C library by replacing the implementation of its operators with our new algorithms, but retaining the APRON API. This means that program analyzers using APRON can directly benefit from our work. We refer to our new version of APRON as OptOctagon. The full implementation in double precision can be found in [1] and is used in our experiments below.

Platform. All of our experiments were carried out on a 3.5 GHz Intel Quad Core i7-4771 Haswell CPU. The sizes of the L1, L2 and L3 caches are 256 KB, 1024 KB and 8192 KB, respectively, and the main memory has 16 GB. Turbo boost was disabled for consistency of measurements. The library was compiled with gcc 4.8 using the flags -O3 -m64 -march=native.

Analyzers. To evaluate the performance of our library, we used four realistic static analyzers written in different languages. All of these analyzers use the APRON library for octagon analysis. For each analyzer, we report only those benchmarks which had at least 20 variables.

The first analyzer, CPACHECKER(CPA) [6], is a configurable program analyzer used in software verification competitions for various verification tasks. It is written in Java and originally used an outdated and incompatible version of APRON. We modified the source code of the analyzer so that it uses the latest version of APRON. We tested our library on more than 40 benchmarks for this analyzer which meet our criteria, some of these contain more than 10000 lines of code. We report speedups on four benchmarks which are representative of speedups on the remaining benchmarks.

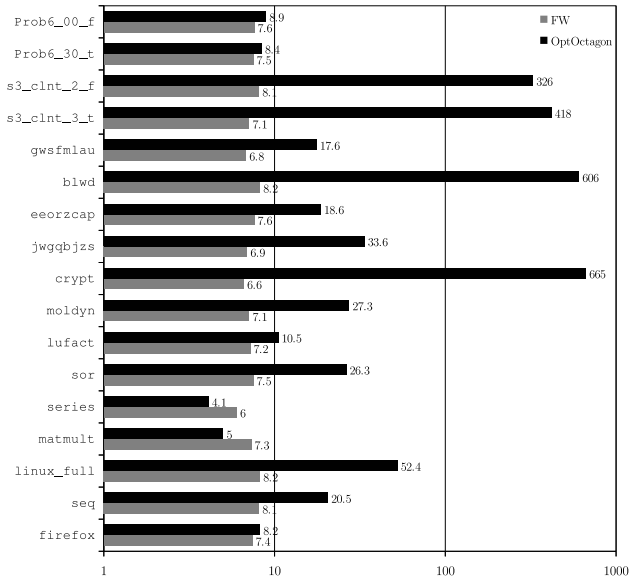


Figure 6: Speedup (in log scale) of our closures over the APRON closure: a) Floyd-Warshall (FW) based closure for dense DBMs, and b) OptOctagon.

The second analyzer, TOUCHBOOST(TB) [8], analyzes event-driven TouchDevelop applications. The analyzer is written in Scala and runs on top of Sample [14]. We report speedups on the four most time consuming benchmarks which time out with APRON after 1 minute.

The third analyzer, DPS [25], analyzes Java programs and introduces synchronization in order to make programs deterministic. The analysis is implemented in Java and uses the Soot [31] framework (it also uses pointer analysis). We report speedups on all six benchmarks for this analyzer.

The fourth analyzer, DIZY [24], computes semantic differences between a program and a patched version of the same program. The analysis is written in C++ and uses LLVM and Clang. We report speedups on three representative benchmarks, two of which `linux_full` and `firefox` are patches from the Linux kernel and the Mozilla Firefox web browser.

6.2 Evaluation

We evaluated the runtime and speedup of our library over APRON at three levels of granularity: 1. The closure operator (a key bottleneck in octagon analysis); 2. End-to-end octagon analysis; and 3. Overall program analysis.

Closure evaluation. We first determine the speedup that our closure operators achieve compared to the existing APRON closure (shown in Fig. 6). The values are computed using the aggregate times that the benchmarks spend in the closure. First, we consider a AVX-vectorized and optimized Floyd-Warshall based implementation (FW) for dense DBMs, derived from Algorithm 1. We do not

Table 2: Closure statistics for benchmarks.

Benchmark	Analyzer	$n_{\min}$	$n_{\max}$	*#closures*
Prob6_00_f	CPA	44	58	4813
Prob6_30_t	CPA	44	58	22170
s3_clnt_2_f	CPA	72	72	708
s3_clnt_3_t	CPA	79	79	715
gwsfmlau	TB	166	186	837
blwd	TB	5	50	24170
eeorzcap	TB	7	93	5398
jwgqbjzs	TB	187	190	1884
crypt	DPS	9	237	861
moldyn	DPS	9	67	5365
lufact	DPS	12	31	142
sor	DPS	16	54	70
series	DPS	8	21	37
matmult	DPS	8	24	10
linux_full	DIZY	1	78	15900
seq	DIZY	1	35	11216
firefox	DIZY	1	24	1061

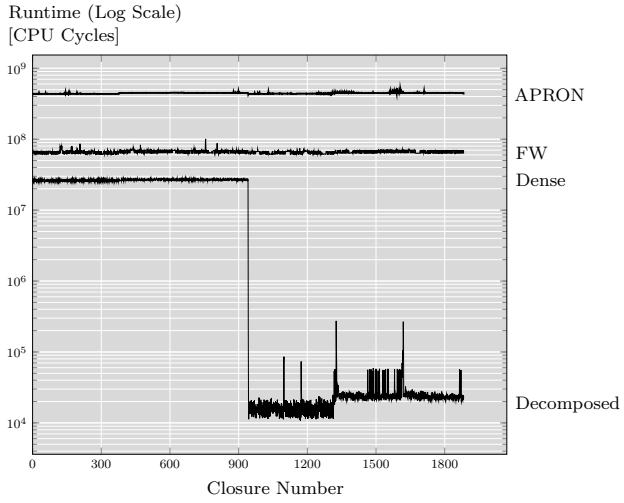


Figure 7: Trace of runtime of octagon closure on jwgqbjz benchmark for APRON, Floyd-Warshall(FW) and OptOctagon.

actually use this code but only show the speedup that one can obtain with processor-specific optimization without reducing the operations count. The speedup is shown in the gray bars and is about 6–8 times. Second, we show the speedup over APRON that our (actually used) OptOctagon closure achieves, which switches between different closures as described in Section 5 and is implemented in a performance-optimized way. It is usually at least as good as FW but often performs around 20 times and sometimes even more than 600 times better than APRON closure. The latter cases obviously benefit considerably from the decomposition.

The number of variables which a DBM encodes varies during analysis. Table 2 shows the number of closure operations (*#closures*) as well as minimum ($n_{\min}$) and maximum ($n_{\max}$) number of variables in DBMs for the closure operator on all the benchmarks. Comparing to Fig. 6, we conclude that the speedup for OptOctagon closure increases with $n_{\max}$ and *#closures*.

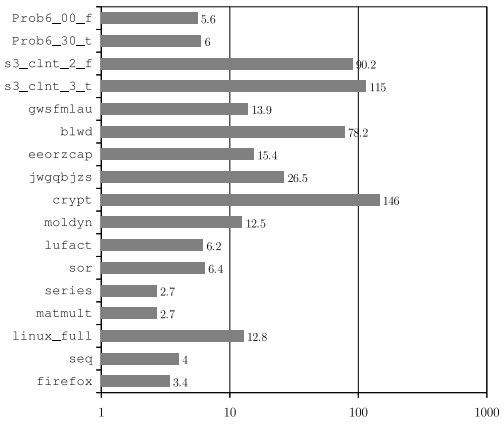


Figure 8: Speedup of Octagon analysis for OptOctagon over APRON on benchmarks (shown in Log Scale).

Fig. 7 shows the runtime of each octagon closure operation during the analysis of the jwgqbjz benchmark. As we can see, the DBMs are dense in the beginning but become sparse due to widening midway through the analysis. FW is 7–8 times faster than APRON closure whereas OptOctagon is an additional 3 times faster than FW for dense DBMs. When the DBMs becomes sparse, the library switches to *Decomposed* type which provides more than a 1000-fold speedup over FW.

Octagon analysis evaluation. Fig. 8 shows the end-to-end speedup for octagon analysis that OptOctagon achieves over APRON on all benchmarks. The best are speedups of 146 times and 115 times on crypt and s3_clnt_3_t. We obtain more than 10 times speedup for 9 of the 17 benchmarks. The speedup is less for smaller benchmarks and the minimum speedup is 2.7 times for series and matmult.

Program analysis evaluation. A static analyzer typically consists of many components besides a numerical abstract domain such as front-end for parsing, other domains (e.g., pointer analysis), etc. Thus, the overall analysis speedup is smaller than the pure octagon speedup. In the future, we believe that some of these components can also be optimized for performance using methods similar to the ones in this paper. Table 3 shows the total end-to-end runtime and speedup of program analysis using OptOctagon instead of APRON on all benchmarks. We also show the percentage of time that analyzers spent in octagon analysis. Octagon analysis is the bottleneck for CPACHECKER and TOUCHBOOST. Thus, we obtain large speedups for all benchmarks corresponding to CPACHECKER and TOUCHBOOST with the maximum being $18.7x$ on the jwgqbjz benchmark. crypt is the only benchmark for DPS having octagon analysis as the main bottleneck. We reduce the program analysis time for crypt by $4x$. The overall speedup for the other DPS analyses is negligible. Similarly, octagon analysis is not the main bottleneck for DIZY. The maximum speedup is 40% on linux_full.

7. Related Work

We already discussed some of the related work throughout the paper. We now briefly mention additional related work. PPL [2] also provides an implementation of the Octagon domain. It has similar data structures and algorithms as APRON so we expect it to have similar performance. The work of [9] presents a new algorithm for reducing the operation count of incremental closure for the Octagon domain. Their work does not handle closure which is the key bot-

Table 3: Speedup of program analysis for OptOctagon over APRON on benchmarks.

Benchmark	Analyzer	APRON		OptOctagon		Speedup
		Time (s)	%oct	Time (s)	%oct	
Prob6_00_f	CPA	29.9	79.4	11.2	38	2.7
Prob6_30_t	CPA	97.5	88.9	26.7	54.5	3.7
s3_clnt_2_f	CPA	7.2	76.4	1.7	3.6	4.2
s3_clnt_3_t	CPA	9	80.8	1.7	3.7	5.3
gwsfmlau	TB	83.5	96.3	8.9	65.2	9.4
blwd	TB	79.1	80.4	16	5	4.9
eeorzcap	TB	89.1	92.6	11.6	46.6	7.7
jwgqbjzs	TB	266	98.5	14.2	69.7	18.7
crypt	DPS	147	77.8	34.7	2	4.2
moldyn	DPS	31.9	17.4	27	2	1.2
lufact	DPS	20	0.3	19.2	0.06	1
sor	DPS	19.2	0.6	19.3	0.1	1
series	DPS	19.7	0.09	19.4	0.03	1
matmult	DPS	19.6	0.03	19.4	0.01	1
linux_full	DIZY	1681	27.5	1244	2.9	1.4
seq	DIZY	155	11.6	129	3.4	1.2
firefox	DIZY	6	13.9	5	4.9	1.2

tleneck, and the scope for overall performance improvements by optimizing only incremental closure is very limited. The work of [5] parallelizes the standard octagon operators on GPUs. We believe that our improved algorithms when implemented on GPUs will bring even larger speedups. The work of [17] decompose matrices in the Polyhedra domain dynamically based on independent components. However, their decomposition is computed separately for each operator and involves expensive matrix transformations thereby creating significant overhead.

8. Conclusion and Future Work

We presented a comprehensive approach for optimizing the Octagon domain, an effective numerical domain for real-world program analysis. Our work shows how to decompose the Octagon matrix online, during analysis, and redesigns the Octagon operators in a way to take advantage of this decomposition. This involves creating new data structures and algorithms as well as dynamically adjusting the Octagon algorithms depending on the sparsity levels. We also showed how to leverage classic numerical code optimizations to further reduce the cost of the expensive closure operator.

We provide a complete implementation of our approach inside the APRON C library. Evaluation on four realistic static analyzers shows massive speedups in the Octagon analysis (over original APRON) as well as significant speedups of the overall static analysis. Based on these results, we believe that static analyzers using the Octagon numerical domain will see immediate benefits from using our new implementation. As future work, we believe the approach presented here can be applied to other domains [12], [10], [27].

References

[1] Optoctagon. https://github.com/eth-srl/OptOctagon.

[2] 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(12):3 – 21, 2008.

[3] R. Bagnara, P. Hill, and E. Zaffanella. Weakly-relational shapes for numeric abstractions: improved algorithms and proofs of correctness. *Formal Methods in System Design (FMSD)*, 35(3):279–323, 2009.

[4] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.

[5] F. Banterle and R. Giacobazzi. A fast implementation of the octagon abstract domain on graphics hardware. In *Proc. International Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 315–335. Springer, 2007.

[6] D. Beyer and M. Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003.

[8] L. Brutschy, P. Ferrara, and P. Müller. Static analysis for independent app developers. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 847–860, 2014.

[9] A. Chawdhary, E. Robbins, and A. King. Simple and efficient algorithms for octagons. In *Programming Languages and Systems*, volume 8858 of *Lecture Notes in Computer Science*, pages 296–313. Springer, 2014.

[10] R. Claris and J. Cortadella. The octahedron abstract domain. In *Proc. International Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2004.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–96, 1978.

[13] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *Proc. International Conference on Formal Verification of Object-oriented Software*, pages 10–30, 2011.

[14] P. Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 8318 of *Lecture Notes in Computer Science*, pages 302–321. Springer, 2014.

[15] R. W. Floyd. Algorithm 97: Shortest path. *Communications ACM*, 5 (6):345–, June 1962.

[16] K. Goto and R. Van De Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1):1–14, 2008.

[17] N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design (FMSD)*, 29(1):79–95, 2006.

[18] S.-C. Han, F. Franchetti, and M. Püschel. Program generation for the all-pairs shortest path problem. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 222–232, 2006.

[19] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[20] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2009.

[21] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Proc. International Static Analysis Symposium (SAS)*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2000.

[22] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *ACM Symposium on Applied Computing*, pages 184–188, 2008.

[23] A. Miné. The octagon abstract domain. *Higher Order and Symbolic Computation*, 19(1):31–100, 2006.

[24] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *Proc. International Static Analysis Symposium (SAS)*, volume 7935 of *Lecture Notes in Computer Science*, pages 238–258. Springer, 2013.

[25] V. Raychev, M. T. Vechev, and E. Yahav. Automatic synthesis of deterministic concurrency. In *Proc. International Static Analysis Symposium (SAS)*, volume 7935 of *Lecture Notes in Computer Science*, pages 283–303. Springer, 2013.

[26] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[27] A. Simon and A. King. The two variable per inequality abstract domain. *Higher Order and Symbolic Computation*, 23(1):87–143, 2010.

[28] A. Toubhans, B. E. Chang, and X. Rival. Reduced product combination of abstract domains for shapes. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7737 of *Lecture Notes in Computer Science*, pages 375–395. Springer, 2013.

[29] C. Urban and A. Miné. An abstract domain to infer ordinal-valued ranking functions. In *Programming Languages and Systems - 23rd European Symposium on Programming (ESOP)*, volume 8410 of *Lecture Notes in Computer Science*, pages 412–431. Springer, 2014.

[30] C. Urban and A. Miné. A decision tree abstract domain for proving conditional termination. In *Proc. International Static Analysis Symposium (SAS)*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014.

[31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - A Java bytecode optimization framework. In *Proc. Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135, 1999.

[32] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded c programs. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 231–242, 2004.