Incremental Transitive Closure for Zonal Abstract Domain

Kenny Ballou[0000−0002−6032−474X] and Elena Sherman[0000−0003−4522−9725]

Boise State University{kennyballou,elenasherman}@boisestate.edu

Abstract. The Zonal numerical domain is an efficient, weakly-relational abstract domain in static analysis by abstract interpretation. Compared to the Interval domain, the Zonal domain is capable of discovering weak relations between two program variables. To reason about Zonal states, it is imperative that they are transformed into a canonical closed form. This task is accomplished through the transitive closure operation commonly implemented as the all-pairs shortest path algorithm, with $O(n^3)$ complexity, where n is the number of program variables.

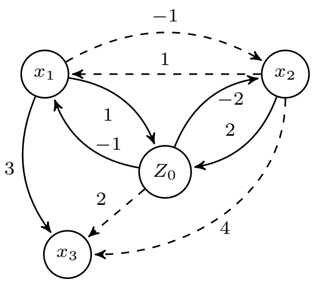
In this work, we explore the closed form of Zonal states in the context of a data-flow analysis framework. Also, we present an incremental transitive closure algorithm that preserves a closed form of an updated Zonal state. The algorithm reduces the overall analysis complexity to $O(n^2)$. We evaluate our approach by performing intra-procedural Zonal analysis on 63 real-world programs. The results show an improvement in runtime, especially on large programs. For example, an hour-long analyzer run with the traditional Zonal implementation has been reduced to a minute with the proposed incremental Zonal variant.

1 Introduction

Abstract interpretation (AI) [4] is an essential technique for supporting various software engineering and programming languages tasks. Used in the context of data-flow analysis framework [7], AI assists a static analyzer with computing invariants over program variables. Then areas such as program verification [2,16] or compiler optimization [6,1] exploit these invariants to accomplish their tasks.

To capture the abstract semantics of a program, AI employs abstract numerical domains, which vary in their expressive power. The Interval domain abstracts program variables into a single continuous interval. Relational numerical domains, such as the Zone and Octagon domain [10,9], are more expressive because they represent relations between program variables. However, the expressiveness of relational numerical domains comes with a higher runtime cost [9]. The Zonal domain is the most efficient among relational domains, but it still timeouts on large programs because of its cubic complexity in terms of program variables [10]. This complexity comes from the transitive closure algorithm for computing canonical representations for Zonal states, which is imperative when comparing Zonal states or identifying infeasible states.

In this work, we investigate the full closure property of Zonal abstract states in the context of a data-flow static analysis framework. While previous work [10]



(a) Graph representation of a Zonal state. Dashed edges are implicit relations.

$$\begin{array}{c|cccc} & Z_0 & x_1 & x_2 & x_3 \\ \hline Z_0 & 0 & -1 & -2 & \boxed{2} \\ x_1 & 1 & 0 & \boxed{-1} & 3 \\ x_2 & 2 & \boxed{1} & 0 & \boxed{4} \\ x_3 & \top & \top & \top & 0 \end{array}$$

(b) Difference Bounded Matrix encoding of the graph on the left. Dashed boxes denote implicit relations.

Fig. 1: A directed graph for $x_1 = 1$, $x_2 = 2$ and $x_1 - x_3 \leq 3$ and the corresponding difference bounded matrix encoding .

defines transfer functions on the Zonal domain regardless of its full closure property, we observed that the fixpoint algorithm frequently compares the Zonal states from the current and previous iterations, which requires invocation of the closure algorithm. This observation prompted us to explore whether the efficiency of transfer functions for fully closed Zonal states can be improved.

We evaluated three implementations of Zonal states in an intra-procedural branch-sensitive data-flow analysis framework on 63 real-world programs. We constructed our experiment to answer the following two research questions:
RQ1: In the context of data-flow analysis, does the propagation of fully closed Zonal states improve runtime efficiency of the analysis?
RQ2: In the context of data-flow analysis, is the proposed incremental transitive closure algorithm more efficient than a conventional closure implementation?

Before we answer these questions in Section 4, we first present necessary background on Zonal abstract domain and then proposed algorithm in Section 3. We conclude with the paper's summary and directions for future work.

2 Zonal Abstract Domain

The Zonal [10] abstract domain is a weakly-relational domain that includes only constraints of the form $x - y \leq c$, where x and y are program variables and c is a numerical constant, in our case an integer. To represent constraints of the form $x \leq c$ in the above canonical form, a special "zero" variable, Z_0, is introduced. Since its value is always 0, the constraint becomes $x - Z_0 \leq c$. The set of linear inequalities represents a bounding region of program variables' possible values.
Representation. The advantages of Zonal domain are that its state can be efficiently represented as a directed graph, and operations on states reduce to graph operations. Figure 1a gives a graph example and Figure 1b the corresponding encoding as a difference-bounded matrix (DBM) [5]. Here, a constraint $x - y \leq c$, is an edge with weight c from the source node x and the target node y. The constraints encoded in Figure 1a (in solid lines) are $x_1 = 1$, i.e., $x_1 - Z_0 \leq 1$ and $Z_0 - x_1 \leq -1$, $x_2 = 2$ and $x_1 - x_3 \leq 3$. Dashed lines represent implicit relations,

Algorithm 1 Forget operation for a traditional Zonal state

```
function CloseAndForget($k$)
    for $i = 0$ to $N$ do                                        ▷ Close connected paths
        for $j = 0$ to $N$ do
            if $(i \neq j \wedge j \neq k)$ then
                $M_{ij} \leftarrow \min(M_{ij}, M_{ik} + M_{jk})$
            end if
        end for
    end for
    for $i = 0$ to $N$ do                                        ▷ Forget constraints connected to $k$
        if $i \neq k$ then
            $M_{ik} \leftarrow \top$
            $M_{ki} \leftarrow \top$
        end if
    end for
end function
```

while the absence of edges indicate unbounded relations between variables. Thus, no edge from x_3 to x_1 indicates the unbounded relation $x_3 - x_1 \leq +\infty$.

The DBM representation places source nodes in rows and target nodes in columns in the same order, and weights between them are elements of the matrix, e.g., the x_1 row and the x_3 column represents the relationship $x_1 - x_3 \leq 3$, and $\top$ values indicate unbounded relations. The values in dashed boxes are implicit relations that are computed by a transitive closure algorithm.

Canonical form. To efficiently compare two Zonal states using their DBM encoding, and perform other essential operations used in a data-flow framework (e.g, `intersection`, `least-upper bound`), it is essential that Zonal states are in the same canonical representation. In the previous example, the set of constraints with solid lines and the same set augmented with implicit constraints (dashed lines) describe the identical bounded region, yet their DBMs are different. Miné [10] proposed a canonical form by transitively closing the set of constraints in a Zonal state. That is, the canonical form where all constraints are explicit, no additional constraints can be inferred. This form is often called *fully closed*.

Essentially, the transitive closure adds implicit constraints, but also tightens the constraints represented by the DBM. Thus, given a DBM M, the transitive closure of M with $n = |M|$ yields the following property: $\forall i, j, k \in \{0, 1, \ldots n\}$, $m_{ij} \leq m_{ik} + m_{kj}$ on elements of M. To transform M into this canonical form, researchers commonly use an all-pairs shortest path algorithm, such as the Floyd-Warshall algorithm [3]. Unfortunately, it has $\Theta(n^3)$ complexity. In fact, this algorithm is primarily the reason for Zonal domain analysis has $O(n^3)$ complexity [10].

Operations. A transfer function of a Zonal state interprets semantics of a statement in terms of removal of existing constraints, i.e., `forget` operation, and addition of a new constraint, i.e., `add` operation. The `add` operation only requires updating a single element of the state's DBM.

Algorithm 2 Incremental Closure Algorithm

```
function IncrementalClosure($s$, $t$, $c$, $M$)
    $N \leftarrow \text{length}(M)$, $W \leftarrow \{t\}$
    if addConstraint($s$, $t$, $c$) then
        for $i = 0$ to $N$ do
            if addConstraint($s$, $i$, $M_{st} + M_{ti}$) then
                $W \leftarrow W \cup \{i\}$
            end if
        end for
        for $i = 0$ to $N$ do
            for $w \in W$ do
                addConstraint($i, w, M_{is} + M_{sw}$)
            end for
        end for
    end if
end function
```

However, the `forget` operation for traditional Zonal states requires additional care, since removing an edge causes all implicit constraints to also disappear, which results in precision loss. Thus, if in Figure 1a the implicit constraints $x_3 \geq -2$ inferred by $x_1 - x_3 \leq 3$ and $x_1 = 1$ is not made explicit before reassigning x_1 (which leads to removing all incoming and outgoing edges from the x_1 node), then the value of x_3 becomes less restricted. As such, the `forget` operation has an intermediate path closure step (lines 2–8 of Algorithm 1) that discovers all implicit paths through the node marked for removal. Afterwards, the algorithm removes all constraints connected to the removed node (lines 9–14). Note, this operation does not remove the variable, instead it removes the constraints associated with the variable, thus making it unbounded. Algorithm 1 presents pseudocode for `forget` as in previous work [10] and has $O(n^2)$ complexity.

If a data-flow framework propagates fully closed Zonal states, however, then the first part of the algorithm on lines 2-9 becomes unnecessary. Thus, for *closed* Zonal states, the complexity of `forget` operation becomes $O(n)$. This complexity reduction comes with a cost – the framework should transform states to their fully closed form ($O(n^3)$). Although, a data-flow framework already requires *closed* Zonal states to perform state comparisons, feasibility checks, and other operations. Hence, the fully closed property of a Zonal state could eliminate invocation of closure algorithm in the context of the framework.

3 Incremental Closure

Since the data-flow framework favors the fully closed form, we investigate whether we can modify the transfer function's operations `add` and `forget` such that for a given fully closed Zonal state they produce a new, fully closed state.

In this case, a constraint removal through `forget` operation is the same as for `closed` Zonal states. If a state is fully closed, then a removal of an edge maintains such a property since no new inferred constraints could be discovered. But `add` operation requires additional considerations. For Zonal states, we propose a novel incremental closure algorithm, which after adding a constraint, also discovers all minimal constraints that can be inferred through that edge. The algorithm computes edges between the source node's parents and the target node's children.

Algorithm 2 presents the pseudocode for the DBM encoding. The parameters s and t are indices of the closed DBM M for the source and the target nodes, and $c \in \mathbb{Z}$ is the constant. If the added constraint is tighter than the existing one, i.e., `AddConstratint` returns true, then it proceeds to discover new implicit constraints. Then the algorithm constructs a worklist of all children which are affected by the addition of the new constraint (lines 4-8). Using this worklist and the parents of s, the algorithm computes the minimum constraint between all the parents of s and the children of t (lines 9-12). The complexity of the incremental transitive closure algorithm is $O(n^2)$ from the two nested loops on the same lines.

Following is a proof of correctness for Algorithm 2.

Theorem 1. *Given a fully closed Zonal state in DBM encoding and a new constraint with s, t and c parameters, the `IncrementalClosure` algorithm computes a correct, fully-closed DBM.*

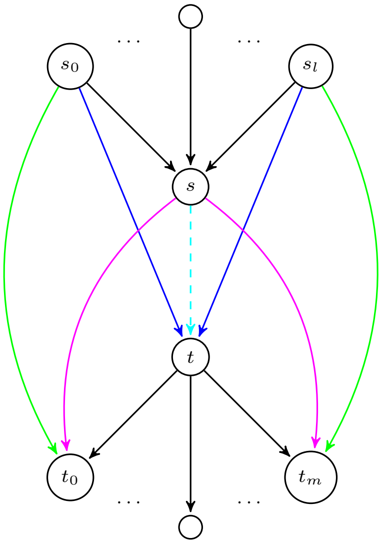


Fig. 2: Example graph representation of M during induction step. The dashed cyan edge represents the additional edge ($s \to t$); blue edges represent parents of s connecting to t; magenta edges represent s connecting to children of t; and green edges represent parents of s connecting to children of t.

Proof. Let V be a finite set of program variables such that $s, t \in V$.

We prove by induction on the number of edges of M, with the full closure property as our induction hypothesis.

Case 1. *Base Case.* Our DBM M has $k = 0$ edges and it is closed. Since $m_{ij} = \top$ $\forall i, j \in \{0, 1, \dots |V|\}$. Therefore, our full closure property, $\forall i, j, h \in \{0, 1, \dots, |V|\}, m_{ij} \leq m_{ih} + m_{hj}$, holds.

Case 2. *Induction.* We assume DBM M with k edges, M is fully closed, and no edge exists between node s and t, i.e., $s - t \leq \top$.

Adding edge $s \rightarrow t$, we have $k + 1$ edges.

Let $S = \text{parent}(s) \cup \{s\}$ and $T = \text{children}(t) \cup \{t\}$. The edges to be recomputed consists of edges from $s_l \rightarrow t_m, \forall s_l \in S$ and $\forall t_m \in T$. We need to show the full closure property holds.

Case (a) *Parents of s connect to t.* This case connects to blue edges in Figure 2.

$\forall s_l \in S$, we connect

$$\overset{*}{m}_{s_lt} \leftarrow \min\left(m_{s_lt}, m_{s_ls} + m_{st}\right)$$

where $\overset{*}{m}_{s_lt}$ is the new edge weight for edge $s_l \rightarrow t$.

Case (b) *s connects to members of T.* This case connects to magenta edges in Figure 2.

$\forall t_m \in T$, we connect

$$\overset{*}{m}_{st_m} \leftarrow \min\left(m_{st_m}, m_{st} + m_{tt_m}\right)$$

where $\overset{*}{m}_{st_m}$ is the new edge weight for edge $s \rightarrow t_m$.

Case (c) *S connects to T.* This case connects to green edges in Figure 2.

$\forall s_l \in S$ and $\forall t_m \in T$, we connect

$$\begin{aligned} \overset{*}{m}_{s_lt_m} &\leftarrow \min\left(m_{s_lt_m},, m_{s_ls} + m_{st} + m_{tt_m}\right) \\ &\leftarrow \min\left(m_{s_lt_m}, m_{s_ls} + \overset{*}{m}_{st_m}\right) \end{aligned}$$

where $\overset{*}{m}_{s_lt_m}$ is the new edge weight for edge $s_l \rightarrow t_m$.

Since either $m_{s_lt_m}$ was already constrained by some h or the addition of edge $s \rightarrow t$ induced a new minimum which was computed above, therefore, $m_{s_lt_m} \leq m_{s_lh} + m_{ht_m}, \forall h \in \{0, 1, \dots |V|\}$ holds. □

4 Evaluations and Results

To evaluate the proposed approaches, we implemented three Zonal branch sensitive intra-procedural analyses: *traditional*, *closed* and *incremental*. We used the Soot (v. `4.2.1.`) data-flow framework [14] that has been extended to support numerical abstract domains [11]. We evaluated these implementations on real-world programs and compared their runtimes. We used the obtained data to answer our two research questions.

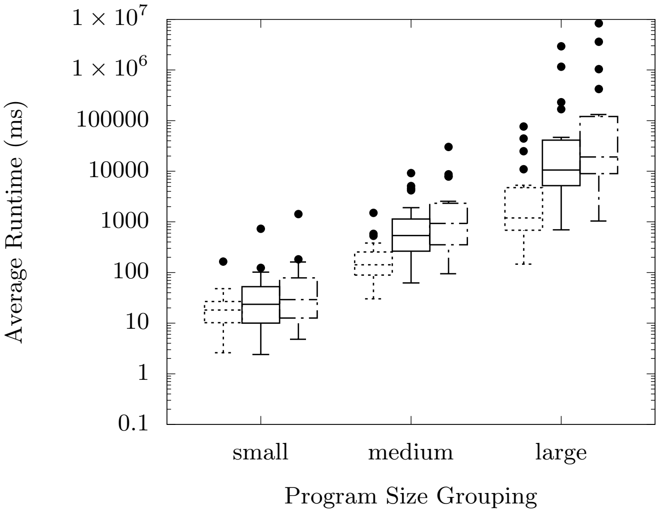


Fig. 3: Average runtime grouped by program sizes. Dotted boxes – *incremental*, solid boxes – *closed*, and dashed – *traditional* Zonal abstract domains.

Benchmarks. Our benchmark set consists of 63 real-world Java methods with non-trivial number of integer operations [12]. To better evaluate the scalability of Zonal implementations, we partitioned the methods into three (3) groups: "small", "medium", and "large". The median instruction count for "small", "medium", and "large" is 23, 156, 468, respectively.

Environment. To avoid inconsistencies within JVM startup times and other experiment supporting operations, we record only the time each analyzer spends performing fixed-point computations. We run each Zonal implementation five times on each program and use the average time as data points.

Results. Figure 3 shows runtime results of our experiments as box plots (milliseconds, log scale y-axis) for each benchmark group (x-axis). The data shows that *closed* (solid boxes) performs slightly better than *traditional* (dashed boxes) Zonal states. Although both Zonal variants invoke the full closure operation at each statement, *closed* uses the full closure property to avoid invocation of this operation at merge and widening points, and when computing branch feasibility. Moreover, the forget operation for *closed* has a linear complexity, while the same operation for *traditional* has a quadratic complexity.

Thus, these improvements contribute to *closed* outperforming *traditional* implementation. However, after performing a t-test, we found no statistically significant differences for all program sizes for $p \leq 0.05$ (for the large group, runtimes become significantly different at $p \leq 0.07$). This small difference in *closed* over *traditional* is due to the dominating cubic complexity of the transitive closure algorithm. Thus, we see a slight improvement of *closed* Zonal state implementation over *traditional*, which indicates that propagating fully closed Zonal states is more efficient in the context of data-flow analysis.

We observe that *incremental* is more efficient compared to *closed*, especially for the large program group. Also, the growth of *incremental* is less steep than the other two variants, because the former is dominated by the quadratic and the latter by the cubic growth complexity in terms of program variables. The data for large program supports this difference in complexity, where the median runtime for *incremental* is about 10^3 ms, while for *closed* is about 10^4 ms.

T-test analyses show no statistical differences for the small group, but found them for the other two groups. The p value for *incremental* vs. *closed* for the medium group is 0.004 and for the large group is 0.002. Thus, we can conclude that our proposed incremental transitive closure algorithm is more efficient than a conventional closure algorithm in the context of a data-flow framework.

5 Conclusion

In this paper, we analyzed propagation of fully closed Zonal abstract states in the context of a data-flow static analysis framework. In addition, we proposed a novel incremental transitive closure algorithm for the Zonal abstract domain and showed analytically and experimentally that it reduces analysis time by an order of magnitude, especially on larger programs.

The representation of DBMs are borrowed from previous work in the model checking community [5,8,15]. This work may be relevant to applications within model checking techniques that require canonical representation of DBMs. In future work, we intend to extend the incremental closure algorithm to allow for more efficient implementations of other canonical forms besides the fully closed canonical form. For example, a canonical form that eliminates relations between constant values, or a minimal canonical form [8].

Acknowledgments

The work reported here was supported by the U.S. National Science Foundation under award CCF-19-42044.

References

1. Abate, C., Blanco, R., Ciobâcă, c., Durier, A., Garg, D., Hritţcu, C., Patrignani, M., Tanter, E., Thibault, J.: An extended account of trace-relating compiler correctness and secure compilation. ACM Trans. Program. Lang. Syst. **43**(4) (nov 2021). https://doi.org/10.1145/3460860, `https://doi.org/10.1145/3460860`
2. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. p. 196–207. PLDI '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/781131.781153, `https://doi.org/10.1145/781131.781153`

3. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, chap. 26.2. Computer science, McGraw-Hill (2009). https://doi.org/10.1.1.708.9446, https://books.google.com/books?id=aefUBQAAQBAJ
4. Cousot, P., Cousot, R.: 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. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973, http://doi.acm.org/10.1145/512950.512973
5. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. Lecture Notes in Computer Science p. 197–212 (1990). https://doi.org/10.1007/3-540-52148-8_17, http://dx.doi.org/10.1007/3-540-52148-8_17
6. Katz, S.: Program optimization using invariants. IEEE Transactions on Software Engineering **4**(05), 378–389 (sep 1978). https://doi.org/10.1109/TSE.1978.233858
7. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 194–206. POPL '73, ACM, New York, NY, USA (1973). https://doi.org/10.1145/512927.512945, http://doi.acm.org/10.1145/512927.512945
8. Larsen, K., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: Proceedings Real-Time Systems Symposium. pp. 14–24. IEEE Comput. Soc (1997). https://doi.org/10.1109/real.1997.641265, http://dx.doi.org/10.1109/REAL.1997.641265
9. Miné, A.: The octagon abstract domain. Higher Order Symbol. Comput. **19**(1), 31–100 (Mar 2006). https://doi.org/10.1007/s10990-006-8609-1, http://dx.doi.org/10.1007/s10990-006-8609-1
10. Miné, A.: A new numerical abstract domain based on difference-bound matrices. Lecture Notes in Computer Science p. 155–172 (2001). https://doi.org/10.1007/3-540-44978-7_10, http://dx.doi.org/10.1007/3-540-44978-7_10
11. Sherman, E.: Redesigning soot's data-flow analysis framework for abstract interpretation. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops. p. 78–84. ISSTA '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236454.3236506, https://doi.org/10.1145/3236454.3236506
12. Sherman, E., Dwyer, M.B.: Exploiting domain and program structure to synthesize efficient and precise data flow analyses (t). 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (11 2015). https://doi.org/10.1109/ase.2015.41, http://dx.doi.org/10.1109/ASE.2015.41
13. Tange, O.: Gnu parallel 20210722 ('blue unity') (Jul 2021). https://doi.org/10.5281/zenodo.5123056, https://doi.org/10.5281/zenodo.5123056, GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them.
14. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. p. 13. CASCON '99, IBM Press (1999)
15. Yovine, S.: Model checking timed automata. Lecture Notes in Computer Science p. 114–152 (1998). https://doi.org/10.1007/3-540-65193-4_20, http://dx.doi.org/10.1007/3-540-65193-4_20

16. Zhu, H., Magill, S., Jagannathan, S.: A data-driven chc solver. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 707–721. PLDI 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192416, https://doi.org/10.1145/3192366.3192416