muRelBench: MicroBenchmarking for Zonotope Domains

Kenny Ballou$^{[0000-0002-6032-474X]1}$ and Elena Sherman$^{[0000-0003-4522-9725]2}$

[1] California State University San Marcos, kballou@csusm.edu
[2] Boise State University elenasherman@boisestate.edu

Abstract. We present `muRelBench`, a framework for synthetic benchmarks for weakly-relational abstract domains and their operations. This extensible microbenchmarking framework enables researchers to experimentally evaluate proposed algorithms for numerical abstract domains, such as closure, least-upper bound, and forget, enabling them to quickly prototype and validate performance improvements before considering more intensive experimentation. Additionally, the framework provides mechanisms for checking correctness properties for each of the benchmarks to ensure correctness within the synthetic benchmarks.

Keywords: Weakly-Relational Abstract Domains · Zonotopes · Benchmarks · Tests

1 Introduction

Zonotopes [9], relational numerical abstract domains, are widely used in program and system verification using static analysis and model-checking techniques and, recently, found their way into the verification of neural networks [11]. To reason about their computations, verifiers manipulate abstract domains through a predefined set of operations, e.g., Least-Upper Bound (LUB), closure, or forget operators [15]. Such manipulations of abstract states commonly dominate the computation time of a verifier. Consequently, there has been extensive research on improving the efficiency of operations over Zonotopes such as closure [1, 3, 7, 15, 18].

While new algorithms provide their complexity estimates, empirically evaluating their runtimes remains crucial to comprehensively assessing their impact. Commonly, such evaluations are performed in the context of a verifier and its target, e.g., a data-flow analyzer using Zones [13, 14] over a set of programs. However, depending on program structure and semantics [2], one may or may not detect the effect of the new operation over the abstract domain. As such, the question shifts to whether the set of programs are representative or the implementation of the new algorithm is inefficient and requires additional tuning. Because of the complexity of Zonotope states, it is difficult to assess whether a verifier produces states with properties that a novel operation algorithm sufficiently takes advantage.

This problem is known to other research communities such as software engineering and compiler optimization community, which they solve by establishing microbenchmarking frameworks [12]. Microbenchmarking isolates the effects of a specific technique such as a certain optimization on syntactically generated code with desired features. In this work, we introduce `muRelBench`[1], an extensible microbenchmarking framework for Zonotopes that is built on top of the `JMH` [16, 17] profiling tool for Java programs. `muRelBench` eliminates verifier and program dependencies and focuses on specific operations of parameterized Zonotope states.

[1] Available on GitHub: https://github.com/fmsea/muRelBench.

For a given type of Zonotope domain, Z and its operation *ops*, `muRelBench` takes as input a set of predefined parameters for each characteristic of the corresponding Z typed abstract domain. Then the framework exhaustively generates abstract states corresponding to each element of the Cartesian product of those parameters and applies *ops* and correctness checks, if any, within the `JMH` context. Upon the completion of experiments, `muRelBench` writes the runtime results for each abstract domain to a variety of output formats, including Comma-Separated Values (CSV) files or `JSON` files, which researchers can use for further analysis and evaluation.

In its current version, generation of abstract states is parameterized by the number of variables and variable connectivity for the Octagons(Z) [15]. Thus, synthetically generated matrices that encode Octagon states vary in their size and variable relation *density*. `muRelBench` implements two closure operations (*ops*): Full Transitive Closure (using Floyd-Warshall all pairs shortest path [5]) and Chawdhary [3] incremental closure. However, as we describe in the next section, `muRelBench` can be easily extended to different Z and *ops* types.

This microbenchmarking framework has the following three key features: (1) dynamic generation of parameterized abstract states, (2) application of user defined operations on them, and (3) checks to user-defined properties, e.g., pre/post conditions on Zonotope states before and after executing operations. We believe that `muRelBench` will help rapid prototyping of abstract operations and evaluating the efficiency of existing implementations.

In the next section 2, we describe framework details and explain how the framework generates different abstract states. To demonstrate the usefulness of `muRelBench`, in Section 3, we present a case study on runtime data of two closure operators on Octagon states. We conclude the paper with future work on `muRelBench`.

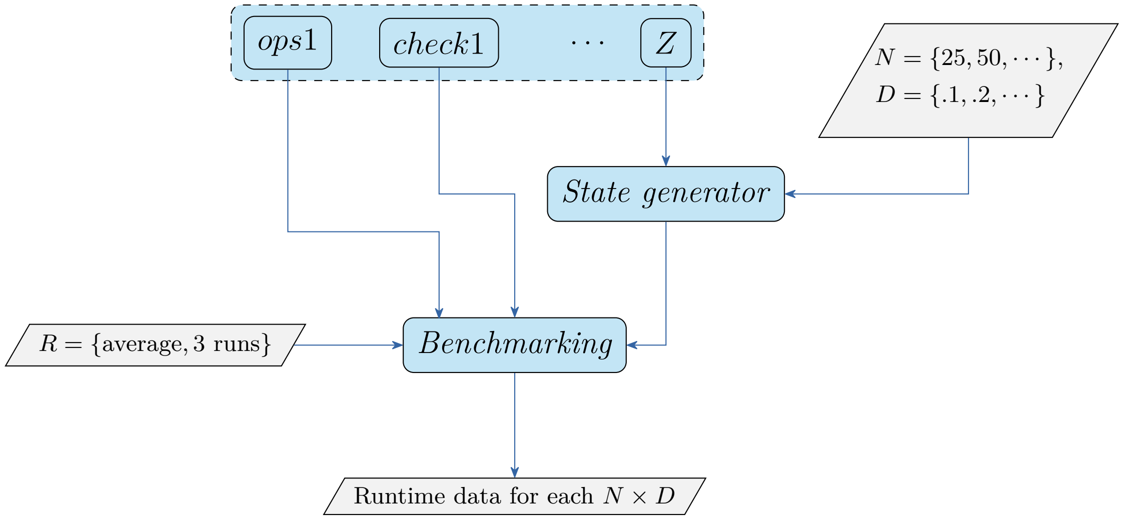
2 `muRelBench` Framework

Figure 1 provides an overview of `muRelBench`'s components. In the dashed, rounded rectangle are user-defined components of an abstract domain type Z, operations, e.g., *ops*1, and property checks of the state after *ops*1 modifies the abstract state. These bindings are defined at compile-time. A *state generator* component takes generation parameters N and D— number of variables and density of the synthetic difference bounded matrix (DBM), respectively, and Z type, and randomly (up to the seed) generates $N \times D$ abstract states.

The *Benchmarking* component takes the generated states and applies *ops*1 state operation and checks the results with *check*1. The component also takes the runtime parameters for `JMH` that defines what type or runtime data to collect and how many times to repeat the experiments. Upon completion, the data is written to the console and, optionally, to an output file.

The framework is implemented in Java and uses interfaces and abstract classes to provide extension points for user-defined components. `JMH` provides a strong foundation for constructing and executing profiling benchmarks whilst minimizing confounding runtime variables such as Java Virtual Machine (JVM) startup, Just-in-Time (JIT) warmup, and Garbage Collection (GC) pauses. Specifically, `muRelBench` defaults to *three* warmup iterations before executing *five* experimental iterations for each benchmark. This way, the code under bench has a chance to JIT compile. We do not specifically tackle the notorious issue of CPU boosting and other dynamic scaling policies that generally plague benchmarks.

The framework currently has extension for Octagon abstract domain, i.e., $Z = Octagon$. The implementation encodes Octagon constraints, which are constraints of the form: $\pm x \pm y \leq c$, where $x, y \in V$ where V is the set of program variables and $c \in \mathbb{I}$, where $\mathbb{I}$ is one of $\mathbb{R}$, $\mathbb{Q}$, or $\mathbb{Z}$. Octagons



$ops1$
$check1$
$\cdots$
Z
$N = \{25, 50, \cdots\}$,
$D = \{.1, .2, \cdots\}$
State generator
$R = \{\text{average}, 3 \text{ runs}\}$
Benchmarking
Runtime data for each $N \times D$

Fig. 1: Component diagram of `muRelBench`, specifying the framework's and user-defined components.

are encoded as a 2-dimensional matrix, e.g., DBM [6] in the `OctagonDifferenceBoundedMatrix` class.

To extend operations over Octagons, users would provide extensions to `OctagonDifference-BoundedMatrix`, overriding various operations with their implementation they wish to test. Furthermore, users provide additional instances of `*Bench`, e.g., `JoinBench`. Similar to `JUnit` [4], the naming follows convention: `muRelBench` automatically includes classes containing the `Bench` suffix.

User extension beyond Octagons It is reasonably straightforward to extend `muRelBench` with additional abstract domains. A user must provide three additional classes: the abstract container type for the domain, e.g., `ZoneDBM` to add Zones [13]; a builder for the new abstract type; and finally, a *state* type which provides a container for the different parameterization sets for `JMH`.

3 Octagons and Closure Operation Case Study

Benchmark Set Up We examine the benefits of `muRelBench` in a case study. The framework randomly generates Octagons, varying the *density* of relations between variables to create a continuum of synthetic instances. This density progression roughly correlates to the different instances of Octagons from real programs. That is, early in analysis, variables have a tendency to have few relations as only few program statements are explored. In the middle of analysis, after exploring several assignment statements, variables become tightly coupled with one another. Finally, after several fixed point iterations and widening operations, islands of connectivity emerge [7, 8, 19]. Furthermore, we also vary the number of variables of the synthetic Octagons to account for different programs sizes.

For this case study, we generate Octagons with 25, 50, and 100 program variables, i.e., 50, 100, and 200 variables using the Octagon variable encoding [15]. For each size, we generate Octagons with $10\% - 90\%$ density, in 10% increments. The Cartesian product of these parameters results in

27 Octagon instances. Furthermore, while other tools such as `Apron` [10] can also generate random, synthetic Octagons, we make a point to only generate *consistent* synthetic Octagons.

Using `JMH`, we default to 3 "warmup" iterations and 5 experimental iterations for each benchmark. Thus, for a single benchmark, the operation under test executes 216 times. However, we do provide options for the user to modify and otherwise specify their own desired warmup and experimental iterations, among other options available via `JMH`.

Case Study In this case study, we chose to evaluate different closure algorithms for Octagon abstract domain. Closure represents a critical operation for static program analysis and abstract interpretation because it provides critical functions: normalization for equality comparisons for data-flow analysis (DFA) [1] and precision benefits for other domain operations such as LUB [14].

Canonicalizing or normalizing Octagon states is a necessary operation because an octagonal bounded region can be represented by infinitely many different Octagons. The closure operation normalizes an Octagon by making explicit implicit edges and minimizing edge weights between variables within the Octagons. In the simplest case, this amounts to computing the all-pairs shortest-path problem for the directed, weighted graph used to represent the Octagon.

There exist several algorithms for computing the all-pairs-shortest-path problem for weighted-directed graphs such as Floyd-Warshall and Bellman-Ford algorithms [5]. While these algorithms are relatively simple and straightforward to implement, their cost can be excessive. Floyd-Warshall, for example has cubic time complexity, $\Theta(n^3)$, where n is the number of variables in the abstract Octagon state.

Chawdhary et al. [3] proposed an incremental closure algorithm for Octagons which uses code motion and hoisting to minimize the number of comparisons required to incrementally close an Octagon. Thus, they were able to reduce the incremental closure, a modified Floyd-Warshall, to $O(20n^2 - 4n)$.

Closure	**Program**	**Mean (ms)**	σ
Floyd-Warshall	Fibonacci	144	32.2
	Loop	46.8	3.1
Chawdhary	Fibonacci	117	5.1
	Loop	49.6	10.3

Table 1: Small programs used to demonstrate performance characteristics of using different closure algorithms.

Clearly, these two algorithms should have a different runtime growth with the increased number of variables. We first examined their result in the context of DFA on two small programs to see if any differences can be detected. Table 1 shows the results of the full-closure algorithm Floyd-Warshall and the Chawdhary et al.'s incremental closure. The data is averaged over five executions and includes the mean runtime for each along with their standard deviation, σ. As the data shows, the results are not entirely conclusive since on the `Loop` program, Floyd-Warshall performed better while Chawdhary runs faster on `Fibonacci`. When we analyzed the properties of the two programs, we discovered that `Fibonacci` algorithm had a maximum of six variables with density of 72% and `Loop` program had two variables with no density, which is purely interval.

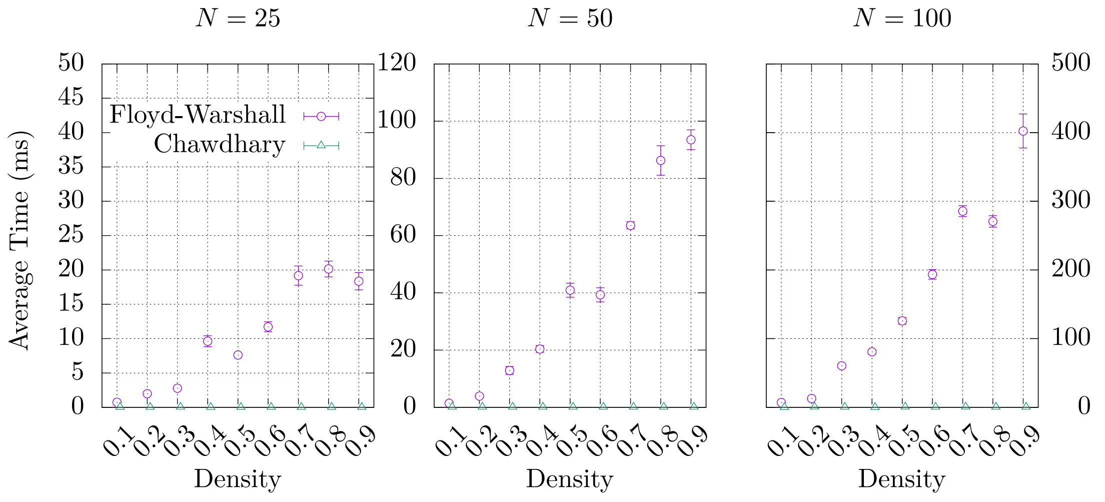


Fig. 2: Plots of microbenchmark results of closure operations, each subplot varies the number of variables, each sample varies the connectivity of program variables.

Plots in Fig. 2 show the results of the comparison of the two closure algorithm on benchmarks that `muRelBench` generates and runs. Each plot presents runtime data for different values of N while varying in density of connections between variables. Using this detailed data, we can discern clear differences between the two algorithms under comparison. Specifically, when the density is small, in each variable instance, the two algorithms seem to perform similarly. However, as soon as the density starts to climb above 30%, the incremental algorithm of Chawdhary et al. clearly computes closure operations more efficiently than that of Floyd-Warshall. Furthermore, variable density shows a significant impact on the runtime for Floyd-Warshall compared to Chawdhary et al.'s, which remains constant.

While it is expected to see a vast performance gap between full closure and incremental closure, we can zoom into the incremental approach and examine two different incremental approaches to closure. Figure 3 shows similar set of plots between the Chawdhary et al. incremental closure algorithm and an incremental closure algorithm similar to the original proposed by Miné [14]. Aside from some expected statistical noise for the small variable size, $N = 25$, these two closure algorithms perform nearly identically. Furthermore, density does not appear to significantly contribute to the runtime of the incremental algorithms under consideration, as was the case for full closure.

It may be tempting to use small programs to quickly validate new algorithm performance, however, such small programs often do not demonstrate realized benefit, as shown in Table 1. The results shown in Fig 2 more acutely capture the performance differences between full closure and an incremental closure. The results of the benchmark would thus encourage further experimentation. However, the results from Fig 3 encourage further algorithmic refinements before more intensive

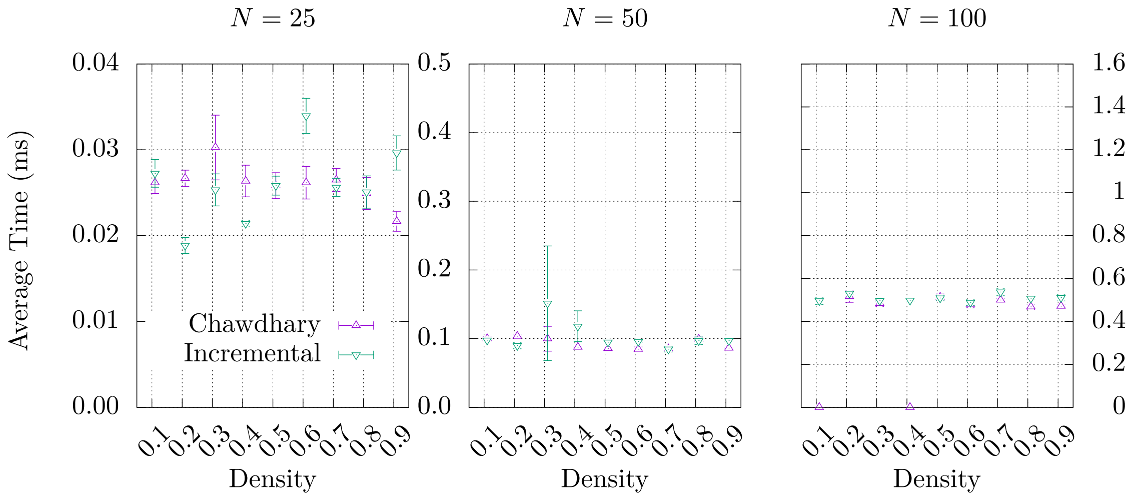


Fig. 3: Plots of microbenchmark results for incremental closure operations. Each subplot again varies by number of variables, each sample varies by connectivity of the program variables.

experimental study. That is, using a tool like `muRelBench` can save time and focus efforts by having a smaller but appropriate set of benchmarks to validate algorithmic improvements[1].

4 Conclusion and Future Work

In this paper, we present the `muRelBench` benchmarking framework to the abstract interpretation research community. This framework offers standardized and uniform support for comparing various operations within Zonotope abstract domains. When developing new algorithms or new abstract domains, a standard set of benchmarks and a common framework to easily test them helps convince the community of their value.

Our framework of generated benchmarks invites many improvements and future work to better situate it for the research community and software engineers at large. For example, we invite contributions of additional algorithms to be added to the suite, so others can use the results in their comparisons. Additionally, more parameters could provide a wider surface area of study for different Zonotope operations.

Acknowledgments

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

[1] The current set of benchmarks were run using GitHub Actions on a default Ubuntu 24.04 Linux-2 instance, which currently only takes approximately 30 minutes.

References

1. Ballou, K., Sherman, E.: Incremental transitive closure for zonal abstract domain. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods. pp. 800–808. Springer International Publishing, Cham (5 2022). https://doi.org/10.1007/978-3-031-06773-0_43, http://dx.doi.org/10.1007/978-3-031-06773-0_43
2. Brunner, R., Dyer, R., Paquin, M., Sherman, E.: Paclab: A program analysis collaboratory. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE '20, ACM (11 2020). https://doi.org/10.1145/3368089.3417936, http://dx.doi.org/10.1145/3368089.3417936
3. Chawdhary, A., Robbins, E., King, A.: Incrementally closing octagons. Formal Methods in System Design **54**(2), 232–277 (1 2018). https://doi.org/10.1007/s10703-017-0314-7, http://dx.doi.org/10.1007/s10703-017-0314-7
4. Contributors, M.: Junit (2024), https://junit.org/
5. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. Computer science, McGraw-Hill (2009). https://doi.org/10.1.1.708.9446, https://books.google.com/books?id=aefUBQAAQBAJ
6. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. Lecture Notes in Computer Science pp. 197–212 (1990). https://doi.org/10.1007/3-540-52148-8_17, http://dx.doi.org/10.1007/3-540-52148-8_17
7. Gange, G., Ma, Z., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: A fresh look at zones and octagons. ACM Transactions on Programming Languages and Systems **43**(3), 1–51 (9 2021). https://doi.org/10.1145/3457885, http://dx.doi.org/10.1145/3457885
8. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Exploiting sparsity in difference-bound matrices. Lecture Notes in Computer Science pp. 189–211 (2016). https://doi.org/10.1007/978-3-662-53413-7_10, http://dx.doi.org/10.1007/978-3-662-53413-7_10
9. Ghorbal, K., Goubault, E., Putot, S.: The zonotope abstract domain taylor1+. Lecture Notes in Computer Science pp. 627–633 (2009). https://doi.org/10.1007/978-3-642-02658-4_47, http://dx.doi.org/10.1007/978-3-642-02658-4_47
10. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. Lecture Notes in Computer Science pp. 661–667 (2009). https://doi.org/10.1007/978-3-642-02658-4_52, http://dx.doi.org/10.1007/978-3-642-02658-4_52
11. Jordan, M., Hayase, J., Dimakis, A., Oh, S.: Zonotope domains for lagrangian neural network verification. Advances in Neural Information Processing Systems **35**, 8400–8413 (2022)
12. Laaber, C., Leitner, P.: An evaluation of open-source software microbenchmark suites for continuous performance assessment. In: Proceedings of the 15th International Conference on Mining Software Repositories. ICSE '18, ACM (5 2018). https://doi.org/10.1145/3196398.3196407, http://dx.doi.org/10.1145/3196398.3196407
13. Miné, A.: A new numerical abstract domain based on difference-bound matrices. Lecture Notes in Computer Science pp. 155–172 (2001). https://doi.org/10.1007/3-540-44978-7_10, http://dx.doi.org/10.1007/3-540-44978-7_10
14. Miné, A.: Weakly Relational Numerical Abstract Domains (12 2004), https://pastel.archives-ouvertes.fr/tel-00136630
15. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (3 2006). https://doi.org/10.1007/s10990-006-8609-1, http://dx.doi.org/10.1007/s10990-006-8609-1
16. Oracle: jmh (2024), https://openjdk.org/projects/code-tools/jmh/
17. Oracle: openjdk/jmh (2024), https://github.com/openjdk/jmh
18. Schwarz, M., Seidl, H.: Octagons revisited. Lecture Notes in Computer Science p. 485–507 (2023). https://doi.org/10.1007/978-3-031-44245-2_21, http://dx.doi.org/10.1007/978-3-031-44245-2_21

19. Singh, G., Püschel, M., Vechev, M.: Making numerical program analysis fast. ACM SIGPLAN Notices **50**(6), 303–313 (8 2015). `https://doi.org/10.1145/2813885.2738000`, `http://dx.doi.org/10.1145/2813885.2738000`