Porting Doop to Soufflé: A Tale of Inter-Engine Portability for Datalog-Based Analyses

Tony Antoniadis Konstantinos Triantafyllou Yannis Smaragdakis

Department of Informatics
University of Athens, 15784, Greece

Abstract

We detail our experience of porting the DOOP static analysis framework to the recently introduced Soufflé Datalog engine. The port addresses the idiosynchrasies of the Datalog dialects involved (w.r.t. the type system, value construction, and fact updates) and differences in the runtime systems (w.r.t. parallelism, transactional execution, and optimization methodologies). The overall porting effort is interesting in many ways: as an instance of the benefits of specifying static analyses declaratively, gaining benefits (e.g., parallelism) from a mere porting to a new runtime system; as a study of the effort required to migrate a substantial Datalog codebase (of several thousand rules) to a different dialect. By exploiting shared-memory parallelism, the Soufflé version of the framework achieves speedups of up to 4x, over already high single-threaded performance.

CCS Concepts • **Theory of computation → Program analysis**

Keywords Points-to analysis, Datalog, Doop

1. Introduction

An important trend in the program analysis literature has been the expression of analyses declaratively, for clearer specification and easier modifiability. The use of the Datalog language for the definition of program analyses has drawn the community's attention in numerous instances [2, 3, 5–7, 7, 9–17, 19, 21]. Datalog is simultaneously a specification logic (with purely declarative semantics) and a programming language. It is a fitting vehicle for expressing the complex recursion in the heart of program analysis: the computational backbone of the language is the definition of recursive relations. Computation in Datalog consists of monotonic logical inferences that apply to produce more facts until fixpoint. A Datalog rule "C(z,x) ⟵ A(x,y), B(y,z)." means that if A(x,y) and B(y,z) are both true, then C(z,x) can be inferred.

The DOOP framework [4] is a prominent instance of the Datalog approach to static analysis. DOOP is an analysis framework for Java bytecode, built around points-to analysis algorithms. It extensively models the diverse semantic aspects of Java and the JVM as Datalog rules. The use of Datalog is not by itself a guarantee of portability. Although the rules are declarative (i.e., evaluation of the rules in any order or reordering the clauses inside a rule yields the same result) the framework's performance crucially depends on optimizing the representation of relations (e.g., defining database indexes). Furthermore, the framework also leverages common Datalog extensions that yield more computational power—e.g., the ability to construct new objects or to do arithmetic.

We present our experience of porting DOOP from its original LogicBlox v.3 Datalog engine to Soufflé [8, 18], a Datalog engine created specifically for program analysis. We discuss the similarities and differences of the two dialects of Datalog (LogiQL for LogicBlox vs. Soufflé Datalog), the limitations and pitfalls of the porting procedure, the architectural differences of the two engines, as well as the difference in optimization methodologies. The main expected benefit is due to parallelism. The LogicBlox v.3 engine underlying DOOP does not take advantage of shared-memory parallelism.[1] Soufflé is built with parallel processing in mind, thus enabling declarative analyses to transparently execute with significant parallel speedup.

The experience we describe is valuable in several ways:

- It is an instance of a port of a substantial declarative codebase, amounting to a roughly 10-person-month effort. It showcases the idiosynchrasies of two different Datalog dialects and optimization methodologies. This can help future porting efforts, either between the precise two Datalog settings we examine or to/from one of the two.
- We present analysis timings for the DaCapo benchmarks to evaluate the performance of full-featured static pointer analyses on the two engines. The result is a validation of the high-level, declarative approach to static analysis implementation and of the use of general analysis engines. The same analysis specification, under a parallel Datalog engine, achieves substantial speedups, up to about 4x.

[1] A new version, v.4 [1], of the LogicBlox engine does emphasize shared-memory parallel execution but DOOP has not been ported to it, due to yet-incomplete support for other essential features, such as complex recursion.

2. Datalog Implementation Differences

The porting effort illustrates major feature differences between the two Datalog engines we consider: the LogicBlox v.3 engine and Soufflé. Major differences exist both in language features (i.e., static and dynamic semantics) and in runtime system implementation (i.e., optimization methodology, performance model, and parallelism). Both kinds of differences had to be overcome in our port, with techniques that we briefly discuss.

2.1 Datalog Dialect Differences

The two Datalog dialects we consider have slightly different syntax. We will not concern ourselves with such minor syntactic details, but will focus on semantic differences between the two languages. In the rest of the text we will leverage the syntax conventions as indicators of the dialect of each code fragment: rules with arrows ("←" and "→") are in the LogicBlox LogiQL dialect; code without arrows (replaced by "**:-**", "**.decl**", or "**.type**") are in Soufflé-Datalog.

Extensional and Intentional Database, Imperative Actions. The LogicBlox system acts as a database that stores persistently the entire current state of computation, i.e., all predicates. Accordingly, LogiQL distinguishes EDB (extensional database) predicates from IDB (intentional database) predicates. The former relations are essentially the input data of the Datalog program, while the latter are computed by (recursive) Datalog rules. The contents of EDB predicates are fixed during the execution of the IDB logic.

The transactional semantics of the LogicBlox system allow it to interleave *declarative evaluation* with *imperative execution*. The language offers a "delta logic" feature, which allows adding or removing facts from EDB relations. Before the transaction performing such an imperative update commits, IDB logic is evaluated incrementally to bring the database again to a consistent state, reflecting the change to the input facts.

In contrast, Soufflé operates as a batch processor of tuples, in a disk-based format. Predicates are read in, computation is performed, and the Datalog program determines which "output" predicates will be re-exported. A program can add facts to both EDB and IDB relations. However, Soufflé does not provide a way to retract tuples from a relation during execution, nor a way to keep a state of computation, incrementally change inputs, and re-evaluate. If further computation is desired, the Datalog program should export all data (which can incur significant cost) and a different program (with compatible type and relation declarations) can re-import it.

The DOOP framework uses delta logic but typically only to add tuples to relations. Therefore, emulating this functionality in Soufflé is easy. Two differences remain. First, the outcome of computation is not an entire database with all (possibly hundreds) of relations, but only the ones that were chosen for export in the program text. Second, DOOP uses delta logic for fact removal in its set-based preprocessing step [20], which simplifies an input program before analysis. This facility is emulated with an export-and-re-import approach.

Type System Differences. There are significant conceptual differences between the type systems of the two Datalog dialects we consider.

- LogiQL is a strongly-typed language. Every value has a unique "principal" type. Soufflé distinguishes symbols and numbers, but otherwise allows constants to be used as values of any compatible type.
- LogiQL has dynamic tagging of types. Every value can be queried (e.g., in a rule body) to retrieve its type (which is unique, per strong typing). In contrast, Soufflé execution carries no dynamic type information.
- LogiQL performs type inference so that most predicates do not need to have the types of their variables declared. In Soufflé all predicates need to have full declarations.

As expected, the above differences have several repercussions on our porting effort. First, we need to explicitly introduce predicates capturing dynamic type information, when this is necessary in the analysis logic. The following snippet of Soufflé Datalog declares three types, HEAPALLOCATION, NUMCONSTANT and VALUE. Based on the type declarations, VALUE can be either a NUMCONSTANT or a HEAPALLOCATION. We also declare the relations ISHEAPALLOCATION, ISNUMCONSTANT and ISVALUE, which are populated by facts of the corresponding type.

```
.type HeapAllocation
.type NumConstant
.type Value = NumConstant | HeapAllocation
.decl isValue(value:Value)
.decl isNumConstant(numConstant:NumConstant)
.decl isHeapAllocation(heap:HeapAllocation)
```

Every time a new value of type HEAPALLOCATION or NUMCONSTANT is produced, it needs to be added to the corresponding IS<TYPE-NAME> predicate in a Soufflé program.[2] This requires care, across several rules that otherwise perform plain computation, to always maintain the necessary dynamic type information for values and the corresponding invariants (e.g., which type is a subtype of another, which two types have mutually exclusive values, etc.). In the LogicQL case, these invariants are maintained implicitly—e.g., when a value of a subtype arises, it is automatically added as a value of the supertype. For the types of our earlier example, the Soufflé program needs explicit rules that fill the ISVALUE relation with tuples of the ISHEAPALLOCATION and ISNUMCONSTANT relations:

[2] In LogiQL, the corresponding predicates would share the name of the type. Soufflé handles types and relations as two separate concepts which share the same namespace. Consequently, a type cannot have the same name as a relation, hence the IS… convention.

```
ISVALUE(numConstant) :- ISNUMCONSTANT(numConstant).
ISVALUE(heap) :- ISHEAPALLOCATION(heap).
```

Furthermore, LogiQL performs type inference at compile time, allowing us to write IDB predicates without declaring the types of their parameters. As an example, we can write the following rule without any declaration for the DECLARINGTYPE predicate. The engine infers that variable *heap* is of type HEAPALLOCATION and variable *type* is of type TYPE, based on the information provided by the joined predicates in the body of the rule.

```
DECLARINGTYPE(heap) = type <-
  ASSIGNHEAPALLOCATION(heap, _, inmethod),
  METHOD:DECLARINGTYPE(inmethod) = type.
```

Soufflé follows a more verbose approach where all relations used in a program must be declared. This has required several tens of extra declarations compared to the original DOOP specification.

The above type-system differences are reflected not just in syntactic shortcuts (e.g., not needing to populate relations, or employing type inference) but also in extra syntactic requirements—neither language can be thought of as a superset of the other. For instance, Soufflé allows more freedom in constructing new objects, since values are not strongly typed. This is reflected well in the construction of context objects for the DOOP static analyses. In the LogiQL code, we need to employ explicit constructor predicates. For instance, the 1-call-site-sensitive analysis of DOOP creates new context objects (from method invocation instructions) via the CONTEXTFROMREALCONTEXT constructor. The constructor is employed to invent "existential" values in the head of a rule and these values can only be used in other head predicates. A standard rule handling virtual method calls appears below.

```
CONTEXTFROMREALCONTEXT(invocation) = ctx ->
  CONTEXT(ctx), METHODINVOCATION(invocation).
lang:constructor('CONTEXTFROMREALCONTEXT).

CONTEXTFROMREALCONTEXT(invocation) = calleeCtx
CALLGRAPHEDGE(callerCtx, invocation, calleeCtx, tomethod),
VARPOINTSTO(hctx, value, calleeCtx, this) <-
  VARPOINTSTO(hctx, value, callerCtx, base),
  VIRTUALMETHODINVOCATIONBASE(invocation, base),
  VALUE:TYPE(value) = valuetype, THISVAR(tomethod) = this,
  RESOLVEINVOCATION(valuetype, invocation) = tomethod.
```

In contrast, Soufflé can construct record objects at will, with no restrictions (as if an infinite universe of them is implied). For the above example, we declare the CONTEXT type to be a record holding a METHODINVOCATION and declare the ISCONTEXT relation storing facts of type CONTEXT. In the body of our rule we freely invent a record value holding the method invocation, as the callee context to be used in the head of the rule (as argument to the VARPOINTSTO and CALLGRAPHEDGE relations). Per our earlier discussion, we explicitly add the newly created callee context to the ISCONTEXT relation.

```
.type CONTEXT[invocation:MethodInvocation]
.decl ISCONTEXT(context:Context)

ISCONTEXT(newCtx),
CALLGRAPHEDGE(callerCtx, invocation, newCtx, tomethod),
VARPOINTSTO(hctx, value, newCtx, this) :-
  VIRTUALMETHODINVOCATIONBASE(invocation, base),
  VARPOINTSTO(hctx, value, callerCtx, base),
  VALUE:TYPE(value, valuetype),
  RESOLVEINVOCATION(valuetype, invocation, tomethod),
  THISVAR(tomethod, this),
  newCtx = [invocation].
```

This flexibility of the Soufflé specification is profitable, for instance, when two types are isomorphic. A context for local variables and a context for heap objects may both consist of just an invocation site (for a 1-call-site-sensitive+heap analysis). In LogiQL, to create one kind of context from the other, we need to unwrap the context object into an invocation-site, and use a second constructor to re-wrap it into a value of the intended type. In Soufflé a mere assignment of values is enough.

Functions and Constraints. There are other differences of the two dialects that afford software engineering convenience, yet do not otherwise impact the code base. Principal among these is the ability to declare, in LogiQL, that a relation is a function—see, e.g., **METHOD:DECLARINGTYPE** and **VALUE:TYPE** in earlier code fragments. The language will detect at run-time any attempt to violate the functional constraint, i.e., to map two values for the same key. This is a major facility for catching errors in LogiQL. Furthermore, this is just an instance of a general mechanism for statically-defined constraints (much like assertions with logical quantifiers) that are checked dynamically to catch errors. Good software engineering practices can minimize the impact of the lack of such language features.

Overall. The two code bases are of similar size, with each burdened by different overheads. In terms of development, the LogicBlox version is rather friendlier due to conveniences such as functional predicates.

2.2 Execution Model and Runtime System

The optimization methodologies and overall execution of the two engines are strikingly different.

Cost Model. Perhaps the greatest difference between the two settings of our port is not in the design of the language dialects but in the execution model that an application programmer should have in mind.

- The LogicBlox engine performs full query optimization. Thus, the user does not need to worry about the order of clauses in the body of a rule. The engine considers *dynamically*, separately for every application instance of a rule, the sizes of the relations involved, and picks a good order for joining the underlying tables. In contrast, in Soufflé the programmer needs to manually decide on

the join order of predicates in a rule body. This manual decision is reflected in the order of clauses in the program text as well as in explicit **.plan** directives for every rule. In addition to being explicit, this join ordering is *static*, applying to all firings of a rule.

- The Soufflé engine automatically optimizes the creation and maintenance of indexes for all relations. The programmer does not need to worry about the order of variables in a predicate, or any other indexing policy. In contrast, in the LogicBlox system, the programmer needs to be aware of the indexing discipline, to reorder variables in a predicate's definition, or to introduce intermediate predicates that encode identical information with different indexing.

The above differences have a significant impact on the DOOP code for the two settings. The methodology for optimizing (via better indexing) DOOP rules for the LogicBlox engine has been described in detail in past work [4]. Effectively, predicates often get their variables reordered, or intermediate predicates are introduced, for better indexing.

The Soufflé burden on the programmer is similar but for different cases. The lack of a query optimizer makes the programmer's ordering of clauses in a rule body crucial for performance. This ordering determines the looping structure in the compiled C++ program. What we try to avoid is iterating over large relations in order to join them with smaller relations. In general we want the outer for-loops to iterate over the smallest relations.

Soufflé also provides the **.plan** directive, for the programmer to specify his/her own query schedule based on different instances of a rule during Datalog *semi-naive* evaluation. Semi-naive evaluation is the standard incrementality strategy for efficient execution of recursive Datalog programs: it executes recursive rules efficiently by considering the "deltas" of each predicate, i.e., the tuples that have been newly produced. With the **.plan** directive programmers can specify their own query schedules based on their predictions for the sizes of relations at run time. Consider the following rule:

```
VARPOINTSTO(hctx, value, ctx, to) :-
  LOADHEAPINSTANCEFIELD(ctx, to, sig, bhctx, baseval),
  INSTANCEFIELDPOINTSTO(hctx, value, sig, bhctx, baseval).
.plan 1:(2,1)
```

The default *plan* is *plan 0*. Based on it, Soufflé will iterate over the LOADHEAPINSTANCEFIELD relation in an outer `for` loop and over the INSTANCEFIELDPOINTSTO relation in an inner `for`, because we expect the deltas of the former to be much smaller in size than the INSTANCEFIELDPOINTSTO relation. The *plan* specified by the programmer, *plan 1*, reverses the order in cases where there is a delta of the INSTANCEFIELDPOINTSTO relation. Essentially, *plan 1* is based on the expectation that the size of the deltas of the INSTANCEFIELDPOINTSTO relation are much smaller in size than LOADHEAPINSTANCEFIELD relation. So Soufflé will evaluate VARPOINTSTO as follows:

$$\Delta\text{VARPOINTSTO} = \Delta\text{LOADHEAPINSTANCEFIELD} \bowtie \text{INSTANCEFIELDPOINTSTO} \cup \Delta\text{INSTANCEFIELDPOINTSTO} \bowtie \text{LOADHEAPINSTANCEFIELD}$$

The lack of a dynamic query optimizer means that we had to resort to newly-introduced intermediate predicates in order to make the analysis efficient for all different context-sensitivity parameterizations of DOOP rules. For instance, consider the rule below (simplified from its original form):

```
VARPOINTSTO(hctx, value, ctx, var) :-
  ASSIGNCONTEXTINSENSITIVEALLOC(value, var, meth),
  REACHABLECONTEXT(ctx, meth),
  IMMUTABLEHCONTEXTFROMCONTEXT(ctx, hctx).
```

The question is whether to iterate over REACHABLECONTEXT before iterating over ASSIGNCONTEXTINSENSITIVEALLOC. The former relation is recursively defined (i.e., has a delta version in semi-naive evaluation) while the latter is not. For a context-insensitive analysis, it is better to iterate over REACHABLECONTEXT first: it is a small predicate, with a unique *ctx* and all the *meth* values that are progressively found reachable. For a context-sensitive analysis, however, the converse is true: the delta predicate of REACHABLECONTEXT can be huge. We resolve such dilemmas by introducing context-insensitive projections of relations (e.g., OPTREACHABLE(*meth*)) and using these (logically redundant) clauses for optimal iteration ordering:

```
VARPOINTSTO(hctx, value, ctx, var) :-
  OPTREACHABLE(meth),
  ASSIGNCONTEXTINSENSITIVEALLOC(value, var, meth),
  REACHABLECONTEXT(ctx, meth),
  IMMUTABLEHCONTEXTFROMCONTEXT(ctx, hctx).
```

A problem with Soufflé's **.plan** directives is that they are brittle. Removing rules results in a change of the remaining rules' stratification, changing which rules are in the same recursive cycle and, thus, which are joined in delta form instead of being fully computed. This renders **.plan** directives invalid. Thus, experimentation with adding and removing rules is rendered harder.

Runtime System. A major difference in the two execution engines is that Soufflé is designed with shared-memory parallelism in mind, whereas LogicBlox v.3 is not. Multithreading can speed up rule evaluation, since relation joins are explicitly parallel. Whether this is profitable depends on the environment. For instance, if one tries to analyze multiple programs simultaneously, the workload has high inherent parallelism, therefore speeding up a single analysis is unwise—it is better to perform a single-threaded analysis of each target, and run all such analyses in parallel. However, when the analysis target is a single large codebase, it is highly desirable to be able to speed up the task with multithreading.

3. Evaluation

We evaluate the performance of our Soufflé port of DOOP over points-to analyses of the DaCapo-2006 benchmarks.

We have concentrated on providing, for the analyses we present, identical analysis results, full language-feature support (modulo reflection, whose implementation is in progress), and highly-optimized rules.

Setup. All experiments are on a dual-Intel Xeon E5-2687W v4 3.00GHz CPU machine. There is 24-way true hardware parallelism, or 48-way hardware parallelism with simultaneous multithreading (*hyperthreading*). The latter is unlikely to benefit this computation-heavy workload.

Discussion. Figures 1 to 3 show timings (single-run, therefore with some noise) for a context-insensitive, a 1-call-site-sensitive, and a 2-object-sensitive+heap points-to analysis. Timings exclude fact-generation (pre-processing) times and other constant overheads. Although these analyses typically finish in a couple of minutes, there are a few longer-running instances, and several clear trends.

- Soufflé single-threaded performance is about 2x faster than LogicBlox if compilation time is excluded. With the addition of compilation time, it becomes slightly slower, up to about 1.7x. However, compilation time is relatively constant (so it matters less in large analyses) and (importantly!) compiled analyses can be reused for different target programs, with very minor modification. Therefore, it depends on one's setting whether compilation time is relevant or not.
- As discussed earlier, it depends on the intended client whether the results of the two engines are equivalent: the LogicBlox output is a full database of all (hundreds of) computed relations, that can form the basis for any further queries. The Soufflé output is the handful of relations selected for export. In our case, this is the VARPOINTSTO and CALLGRAPHEDGE relations—the key outputs of a points-to analysis.
- Soufflé exhibits maximum speedup at 4 or 8 threads and often over 2x (up to 3.9x for bloat/2objH). This is impressive, as it comes over already-high single-threaded performance.

Acknowledgments. We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE).

References

[1] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[2] G. Balatsouras and Y. Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *Proc. of the 23rd International Symp. on Static Analysis*, 2016.

[3] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proc. of the 18th International Symp. on Software Testing and Analysis*, 2009.

[4] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, 2009.

[5] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *Proc. of the 30th International Conf. on Software Engineering*, 2008.

[6] S. Guarnieri and B. Livshits. GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[7] E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest: Scalable source code queries with Datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, 2006.

[8] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, 2016.

[9] G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in Java points-to analysis. In *Proc. of the 22nd International Conf. on Compiler Construction*, 2013.

[10] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2013.

[11] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *Proc. of the 28th European Conf. on Object-Oriented Programming*, 2014.

[12] Y. Li, T. Tan, and J. Xue. Effective soundness-guided reflection analysis. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, 2015.

[13] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *Proc. of the 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2011.

[14] B. Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, 2006.

[15] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Proceedings of the 3rd Asian Symp. on Programming Languages and Systems*, 2005.

[16] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2006.

[17] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proc. of the 31st International Conf. on Software Engineering*, 2009.

[18] B. Scholz, H. Jordan, P. Subotic, and T. Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, 2016.

[19] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[20] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis. Set-based pre-processing for points-to analysis. In *Proc. of the 28th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, 2013.

[21] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer. More sound static handling of Java reflection. In *Proc. of the 13th Asian Symp. on Programming Languages and Systems*, 2015.

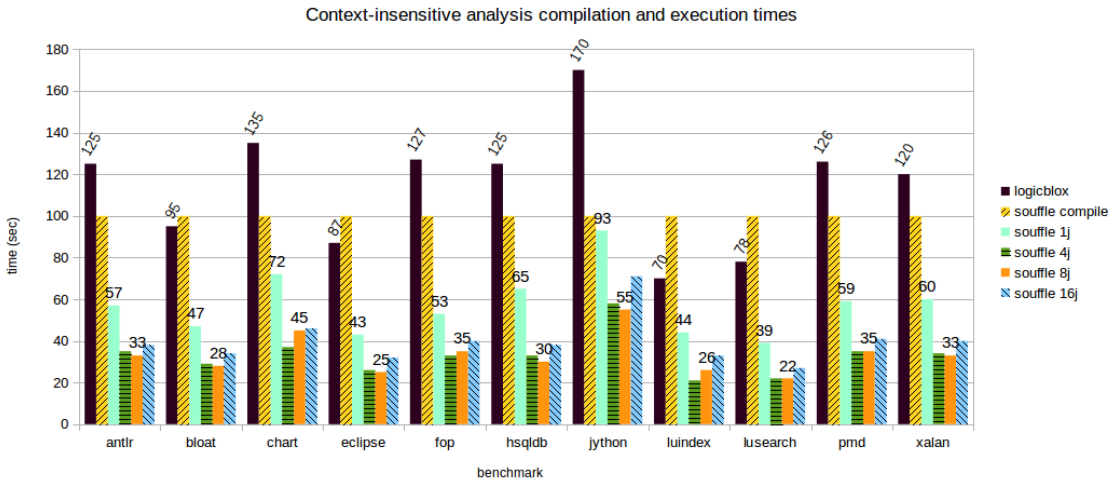

Figure 1: Context-insensitive analysis compilation and execution times for the DaCapo benchmarks.

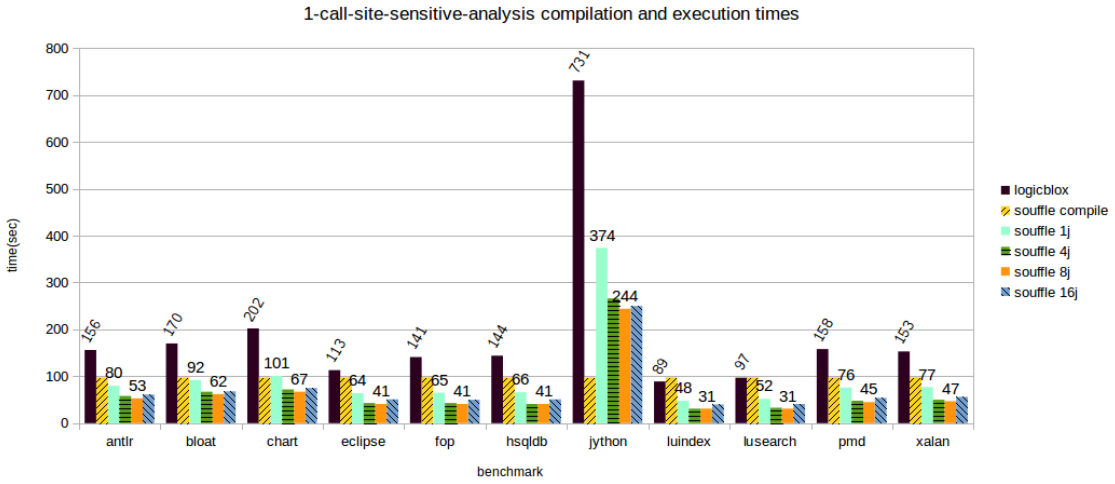

Figure 2: 1-call-site analysis compilation and execution times for the DaCapo benchmarks.

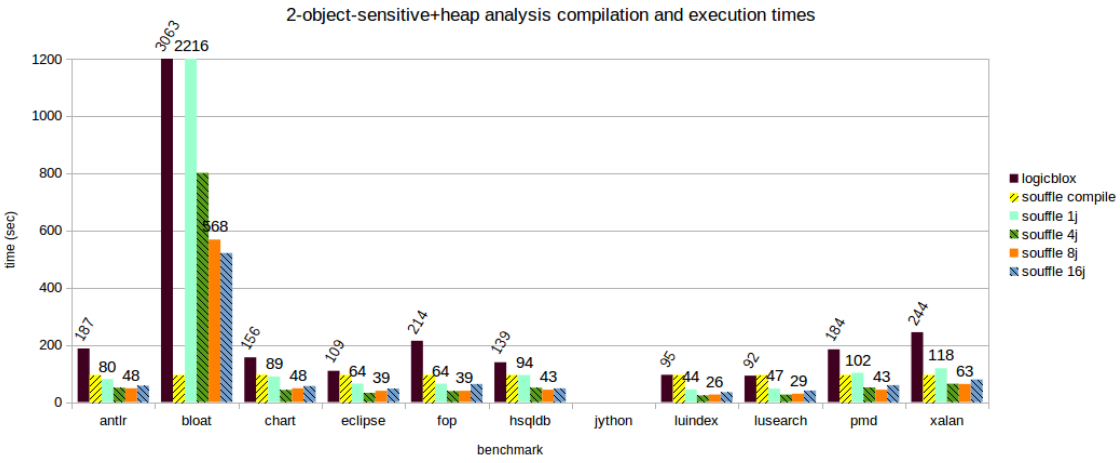

Figure 3: 2-object-sensitive+heap analysis compilation and execution times for the DaCapo benchmarks.