Introspective Analysis: Context-Sensitivity, Across the Board

Yannis Smaragdakis George Kastrinis George Balatsouras

Department of Informatics
University of Athens
{smaragd,gkastrinis,gbalats}@di.uoa.gr

Abstract

Context-sensitivity is the primary approach for adding more precision to a points-to analysis, while hopefully also maintaining scalability. An oft-reported problem with context-sensitive analyses, however, is that they are bi-modal: either the analysis is precise enough that it manipulates only manageable sets of data, and thus scales impressively well, or the analysis gets quickly derailed at the first sign of imprecision and becomes orders-of-magnitude more expensive than would be expected given the program's size. There is currently no approach that makes precise context-sensitive analyses (of any flavor: call-site-, object-, or type-sensitive) scale across the board at a level comparable to that of a context-insensitive analysis. To address this issue, we propose introspective analysis: a technique for uniformly scaling context-sensitive analysis by eliminating its performance-detrimental behavior, at a small precision expense. Introspective analysis consists of a common adaptivity pattern: first perform a context-insensitive analysis, then use the results to selectively refine (i.e., analyze context-sensitively) program elements that will not cause explosion in the running time or space. The technical challenge is to appropriately identify such program elements. We show that a simple but principled approach can be remarkably effective, achieving scalability (often with dramatic speedup) for benchmarks previously completely out-of-reach for deep context-sensitive analyses.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Algorithms, Languages, Performance

Keywords points-to analysis; context-sensitivity; object-sensitivity; type-sensitivity

1. Introduction

Points-to analysis is probably the most common whole-program static analysis, and often serves as a substrate for a variety of high-level program analysis tasks. Points-to analysis computes the set of objects (abstracted as their allocation sites) that a program variable may point to during runtime. The promise, as well as the challenge, of points-to analysis is to yield usefully precise information without sacrificing scalability: the analysis inputs are large and the analysis algorithms are typically quadratic or cubic, but try to maintain near-linear behavior in practice, by exploiting program properties and maintaining precision. Indeed precision and performance often go hand-in-hand in a good points-to analysis algorithm: better algorithms are often found to be both more precise and faster, because smaller points-to sets lead to less work [14].

Context-sensitivity is a common way of pursuing precision and scalability in points-to analysis. It consists of qualifying local variables and objects with context information: the analysis unifies information (e.g., "what objects this method argument can point to") over all possible executions that map to the same context value, while separating executions that map to different contexts. In this way, context-sensitivity attempts to avoid precision loss from merging the behavior of different dynamic program paths. Context-sensitivity comes in many flavors, depending on the kind of context information, such as *call-site-sensitivity* [22, 23], *object-sensitivity* [19, 20], and *type-sensitivity* [24].

An oft-remarked fact about context-sensitivity, however, is that even the best algorithms have a common failure mode when they cannot maintain precision. Past literature reports that "the performance of a [...] deep-context analysis is bimodal" [24]; "context-sensitive analyses have been associated with very large numbers of contexts" [15]; "algorithms completely hit a wall after a few iterations, with the number of tuples exploding exponentially" [16]. Recent published results [12] fail to run a 2-object-sensitive analysis in under 90mins for 2 of 10 DaCapo benchmarks, while 2 more benchmarks take more than 1,000sec, although most other benchmarks of similar or larger size get analyzed in under 200sec.

Thus, when context-sensitivity works, it works formidably, in terms of both precision and performance. When it fails, however, it fails miserably, quickly exploding in complexity. In contrast, context-insensitive analyses uniformly scale well, for the same inputs. Figure 1 vividly demonstrates this phenomenon for the DaCapo benchmarks, analyzed with the Doop framework [2] under a context-insensitive (insens) analysis and a 2-object-sensitive analysis with a context-sensitive heap (2objH). (The chart truncates the analysis time of the longest-running benchmarks. Two of them, hsqldb and jython, timed out after 90mins on a 24GB machine, and would not terminate even for much longer timeouts.) As can be seen, context-insensitive analyses vary relatively little in performance, while context-sensitivity often causes running time (and memory use) to explode.

Faced with this unpredictability of context-sensitivity, a common reaction is to avoid it, favoring context-insensitive analyses, and, consequently, missing significant precision benefits for well-behaved programs. Even worse, for some applications, eschewing expensive context-sensitivity is not an option—a context-insensitive analysis is just not good enough. Reports from industry [4] and academic researchers [3] alike reiterate that precise

PLDI'14, June 9–11, 2014, Edinburgh, United Kingdom..

ACM 978-1-4503-2784-8/14/06. . . $15.00.
http://dx.doi.org/10.1145/2594291.2594320

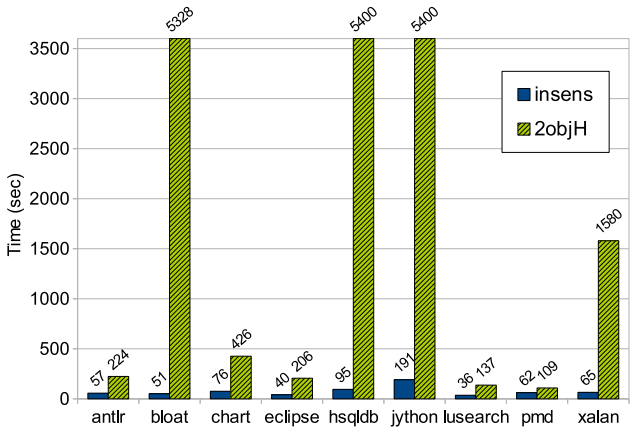

Figure 1. Comparison of running times of context-insensitive analysis vs. 2-object-sensitive with context-sensitive heap. The y-axis is truncated to 1hr for readability.

context-sensitivity is essential for information-flow analysis, taint analysis, and other security analyses.

We can ask ourselves, why does this scalability barrier arise? The core problem is that, for some objects or methods, the points-to information is imprecise enough that more context does not help, while incurring a heavy overhead [24]. Consider a method argument that was found to point to n objects by a less precise analysis. Further analyzing the method in c different contexts (or, equivalently, increasing context depth by 1) will ideally yield n/c points-to facts per context, perfectly splitting the previous n-object points-to set. In the worst case, however, increasing the context depth will result in c copies of n points-to facts each: the extra context depth will not have yielded more precision, but will have multiplied the space and time costs.

The focus of our work is on the detection and prevention of pathological behavior in context-sensitive analyses, with minimal intervention. In this way, we achieve many of the precision benefits of context-sensitivity without sacrificing scalability. It does not seem possible to know in advance (e.g., by identifying syntactic features of the program) which program elements may be responsible for pathological behavior. Nevertheless, we argue that it is possible to identify such elements with a scalable context-insensitive analysis. We introduce the concept of *introspective context-sensitivity*: during a first, context-insensitive, analysis pass, the analysis observes symptoms indicating that the cost may get out of hand for deeper context. This detects exactly the pathology identified above. In its simplest form, the analysis will ask "which program sites currently have points-to information that may grow too large for an extra level of context?" Using a configurable second pass, such sites will be re-analyzed with shallow context, even though the rest of the program will be re-analyzed with a deeper context. Note that this approach adaptively tunes the precision of the analysis for the entire program, and not only for a single goal or single client analysis, as in earlier *refinement-based* [25], *pruning* [16], or *client-driven* [8] techniques—our related work section includes a detailed discussion.

The net outcome of our work is not a "first line of defense" analysis, but an "if all else fails" analysis. Users are still better advised to first use traditional context-sensitive algorithms, in the hope that these will scale well and provide good precision. When this fails, however, we show that we can provide a highly reliable knob so that the user can "dial-in" scalability, to the exact level required. For instance, as seen in Figure 1, a precise 2objH analysis fails to run in under 60mins on a 24GB machine for 3 of our experimental subjects. However, we can get an introspective context-sensitive analysis to scale to all benchmarks in under 12mins, while still gaining significant precision over a context-insensitive analysis. Yet another introspective analysis scales to all but one benchmark in under 20mins, while sacrificing very little precision (keeping about 2/3 of the precision gains of a full 2objH analysis). For call-site-sensitive analyses, the gains are even more pronounced, with several benchmarks exhibiting at least 300% speedups.

Overall, our paper makes the following contributions:

- We offer an approach to refining a context-sensitive analysis while avoiding its worst-case cost. The approach relies on first running a context-insensitive analysis and using its results to inform the application of context-sensitivity. Much of the challenge concerns the question of *how* to use this information, i.e., what heuristics yield good behavior.
- We encode the approach in a simple form, by incremental modifications of a general declarative analysis pattern. Therefore, our approach works on virtually any algorithm expressed in this manner. Our implementation is on the Doop framework [2] and already applies to the over 30 analysis algorithms that the framework has to offer.
- We show experimentally the benefit of introspective context-sensitivity. We quantify the precision loss and scalability gains for different parameter settings and show that there is a dial that users can tune, to select points in this spectrum. Even our high-precision settings are effective in eliminating behavior outliers, showing that introspective context-sensitivity has core value: previously hopeless analyses suddenly become feasible, for little precision loss. We believe that the result is to give confidence that context-sensitive analyses can be used in virtually any setting and not just in the nebulous "when they work well" case.

2. Base Model for Introspective Context-Sensitivity

We demonstrate introspective context-sensitivity via incremental changes to an existing model for context-sensitive, flow-insensitive[1] points-to analysis algorithms [11, 12]. The logical formalism of this model is very close to the core components of our actual analysis implementation. Therefore, the model acts both as background and as the main building block of the refinement logic in later sections. The key element of the model is the ability to use two different kinds of context, on a per-program-site basis.

Preliminaries. Out model captures the core points-to analysis logic, as well as online call-graph construction, as a parametric Datalog program. Datalog rules are monotonic logical inferences that repeatedly apply to infer more facts until fixpoint. Our rules do not use negation in a recursive cycle, or other non-monotonic logic constructs. The result is a declarative specification: the order of evaluation of rules or examination of clauses cannot affect the analysis outcome. The abstract model can be parameterized to yield a context-insensitive Andersen-style [1] analysis, as well as several context-sensitive analyses: call-site-sensitive [22, 23], object-sensitive [20], and type-sensitive [24].

The input language is a representative simplified intermediate language with a) a "new" instruction for allocating an object; b) a "move" instruction for copying between local variables; c) "store" and "load" instructions for writing to the heap (i.e., to object fields);

[1] *Flow-sensitivity* refers to taking into account the intra-procedural control-flow of the program—e.g., that an instruction may always precede another. Flow-sensitivity can be approximated in a flow-insensitive setting by first putting the input in static-single-assignment (SSA) form.

d) a "virtual method call" instruction that calls the method of the appropriate signature defined in the dynamic class of the receiver object. This language models well the Java bytecode representation,[2] but also other high-level intermediate languages. The specification of our points-to analysis as well as the input language are in line with others in the literature [7, 17], although we also integrate elements such as on-the-fly call-graph construction and field-sensitivity.

Specifying the analysis logically as Datalog rules has the advantage that the specification is close to the actual implementation. Datalog has been the basis of several implementations of program analyses, both low-level [2, 13, 21, 30, 31] and high-level [5, 9]. Indeed, the analysis we show is a faithful model of the implementation in the Doop framework [2], upon which our work builds. Our specification of the analysis (Figures 2-3) is an abstraction of the actual implementation in the following ways:

- The implementation has many more rules. It covers the full complexity of Java, including rules for handling reflection, native methods, static fields, string constants, implicit initialization, threads, and a lot more. The Doop implementation contains over 600 rules in the common core of all analyses, and several more rules specific to each analysis, as opposed to the 10 rules we examine here. (Note, however, that these few rules are the most crucial for points-to analysis. They also correspond fairly closely to the algorithms specified in other formalizations of points-to analyses in the literature [18, 24].)
- The implementation also reflects considerations for efficient execution. The most important is that of defining indexes for the key relations of the evaluation. Furthermore, it designates some relations as functions, defines storage models for relations (e.g., how many bits each variable uses), designates intermediate relations as "materialized views" or not, etc. No such considerations are reflected in our model.

High-level structure. The high-level structure of our model is simple: the core analysis is merely the parametric analysis of past work [11, 12] enhanced with the ability to create two different kinds of context—regular or refined. The analysis accepts extra input relations that identify program elements (allocation and call sites) to be analyzed with a refined context. How these input relations are defined is the topic of Section 3.

Figure 2 shows the domain of our analysis (i.e., the different value sets that constitute the space of our computation), its input relations, the intermediate and output relations, as well as four constructor functions, responsible for producing new contexts. Figure 3 shows the points-to analysis and call-graph computation.

Input relations. The input relations correspond to the intermediate language for our analysis. They are logically grouped into relations that represent instructions, relations that represent name-and-type information, and parameterization relations for introspective context-sensitivity. For instance, the ALLOC relation represents every instruction that allocates a new heap object, *heap*, and assigns it to local variable *var* inside method *inMeth*. (Note that every local variable is defined in a unique method, hence the *inMeth* argument is also implied by *var* but is included to simplify later rules.) There are similar input relations for all other instruction types (MOVE, LOAD, STORE, and VCALL).

[2] The Java bytecode language is a stack-based intermediate language, for reasons of compactness. For analysis purposes, however, it is common to translate it into equivalent but more conventional notations, such as the Jimple intermediate language of the Soot framework [28, 29]. It is more accurate to say that our intermediate language is a simplified form of Jimple, rather than a simplified form of the Java bytecode.

V is a set of program variables
H is a set of heap abstractions (i.e., allocation sites)
M is a set of method identifiers
S is a set of method signatures (including name, type signature)
F is a set of fields
I is a set of instructions (mainly used for invocation sites)
T is a set of class types
$\mathbb{N}$ is the set of natural numbers
C is a set of (calling) contexts
HC is a set of heap contexts

ALLOC (*var : V, heap : H, inMeth : M*)	*# var = new ...*
MOVE (*to : V, from : V*)	*# to = from*
LOAD (*to : V, base : V, fld : F*)	*# to = base.fld*
STORE (*base : V, fld : F, from : V*)	*# base.fld = from*
VCALL (*base : V, sig : S, invo : I, inMeth : M*)	*# base.sig(..)*

FORMALARG (*meth : M, i : $\mathbb{N}$, arg : V*)
ACTUALARG (*invo : I, i : $\mathbb{N}$, arg : V*)
FORMALRETURN (*meth : M, ret : V*)
ACTUALRETURN (*invo : I, var : V*)
THISVAR (*meth : M, this : V*)
HEAPTYPE (*heap : H, type : T*)
LOOKUP (*type : T, sig : S, meth : M*)

SITETOREFINE (*invo : I, meth : M*)
OBJECTTOREFINE (*heap : H*)

VARPOINTSTO (*var : V, ctx : C, heap : H, hctx : HC*)
CALLGRAPH (*invo : I, callerCtx : C, meth : M, calleeCtx : C*)
FLDPOINTSTO (*baseH: H, baseHCtx: HC, fld: F, heap: H, hctx: HC*)
INTERPROCASSIGN (*to : V, toCtx : C, from : V, fromCtx : C*)
REACHABLE (*meth : M, ctx : C*)

RECORD (*heap : H, ctx : C*) **= *newHCtx : HC***
MERGE (*heap : H, hctx : HC, invo : I, ctx : C*) **= *newCtx : C***
RECORDREFINED (*heap : H, ctx : C*) **= *newHCtx : HC***
MERGEREFINED (*heap : H, hctx : HC, invo : I, ctx : C*) **= *newCtx : C***

Figure 2. Our domain, input relations, output relations, and constructors of contexts. The input relations are of three kinds: relations encoding program instructions (the kind of instruction is shown in a comment), relations encoding type system and other environment information, and relations that filter which call sites/target methods and which objects should have a different (i.e., more precise) context in introspective context-sensitivity.

Similarly, there are relations that encode type system, symbol table, and program environment information. These are mostly straightforward. For instance, FORMALARG shows which variable is a formal argument of a given method at a certain index (i.e., the i-th argument). LOOKUP matches a method signature to the actual method definition inside a type. HEAPTYPE matches an object to its type, i.e., is a function of its first argument. (Note that we are shortening the term "heap object" to just "heap" and represent heap objects as allocation sites throughout.) ACTUALRETURN is also a function of its first argument (a method invocation site) and returns the local variable at the call site that receives the method call's return value.

Finally, the SITETOREFINE and OBJECTTOREFINE relations are inputs that are used exclusively for the purposes of introspective context-sensitivity. They encode the program points (allocation sites, *heap*, and invocation site/method combinations, *invo*, *meth*) that will employ a different context abstraction from the rest.

```
INTERPROCASSIGN (to, calleeCtx, from, callerCtx) ⟵
  CALLGRAPH (invo, callerCtx, meth, calleeCtx),
  FORMALARG (meth, i, to), ACTUALARG (invo, i, from).

INTERPROCASSIGN (to, callerCtx, from, calleeCtx) ⟵
  CALLGRAPH (invo, callerCtx, meth, calleeCtx),
  FORMALRETURN (meth, from), ACTUALRETURN (invo, to).


RECORD (heap, ctx) = hctx,
VARPOINTSTO (var, ctx, heap, hctx) ⟵
  REACHABLE (meth, ctx), ALLOC (var, heap, meth),
  !OBJECTTOREFINE (heap).

# duplicate rule, for introspective context-sensitivity
RECORDREFINED (heap, ctx) = hctx,
VARPOINTSTO (var, ctx, heap, hctx) ⟵
  REACHABLE (meth, ctx), ALLOC (var, heap, meth),
  OBJECTTOREFINE (heap).

VARPOINTSTO (to, ctx, heap, hctx) ⟵
  MOVE (to, from), VARPOINTSTO (from, ctx, heap, hctx).

VARPOINTSTO (to, toCtx, heap, hctx) ⟵
  INTERPROCASSIGN (to, toCtx, from, fromCtx),
  VARPOINTSTO (from, fromCtx, heap, hctx).

VARPOINTSTO (to, ctx, heap, hctx) ⟵
  LOAD (to, base, fld), VARPOINTSTO (base, ctx, baseH, baseHCtx),
  FLDPOINTSTO (baseH, baseHCtx, fld, heap, hctx).

FLDPOINTSTO (baseH, baseHCtx, fld, heap, hctx) ⟵
  STORE (base, fld, from), VARPOINTSTO (from, ctx, heap, hctx),
  VARPOINTSTO (base, ctx, baseH, baseHCtx).


MERGE (heap, hctx, invo, callerCtx) = calleeCtx,
REACHABLE (toMeth, calleeCtx),
VARPOINTSTO (this, calleeCtx, heap, hctx),
CALLGRAPH (invo, callerCtx, toMeth, calleeCtx) ⟵
  VCALL (base, sig, invo, inMeth), REACHABLE (inMeth, callerCtx),
  VARPOINTSTO (base, callerCtx, heap, hctx),
  HEAPTYPE (heap, heapT), LOOKUP (heapT, sig, toMeth),
  THISVAR (toMeth, this),
  !SITETOREFINE (invo, toMeth).

# duplicate rule, for introspective context-sensitivity
MERGEREFINED (heap, hctx, invo, callerCtx) = calleeCtx,
REACHABLE (toMeth, calleeCtx),
VARPOINTSTO (this, calleeCtx, heap, hctx),
CALLGRAPH (invo, callerCtx, toMeth, calleeCtx) ⟵
  VCALL (base, sig, invo, inMeth), REACHABLE (inMeth, callerCtx),
  VARPOINTSTO (base, callerCtx, heap, hctx),
  HEAPTYPE (heap, heapT), LOOKUP (heapT, sig, toMeth),
  THISVAR (toMeth, this),
  SITETOREFINE (invo, toMeth).
```

Figure 3. Datalog rules for the points-to analysis and call-graph construction.

Computed relations. There are five output or intermediate computed relations: VARPOINTSTO, . . ., REACHABLE.[3] Every occurrence of a method or local variable in computed relations is qualified with a context (i.e., an element of set *C*), while every occurrence of a heap object is qualified with a heap context (i.e., an element of *HC*). The main output relations are VARPOINTSTO and CALLGRAPH, encoding our points-to and call-graph results. The VARPOINTSTO relation links a variable (*var*) to a heap object (*heap*). Other intermediate relations (FLDPOINTSTO, INTERPROCASSIGN, REACHABLE) correspond to standard concepts and are introduced for conciseness and readability.

Constructors for context-sensitivity. The base rules are not concerned with what kind of context-sensitivity is used. The same rules can be used for a context-insensitive analysis (by only ever creating a single context object), for a call-site-sensitive analysis, or for an object-sensitive analysis, for any context depth. These aspects are completely hidden behind constructor functions **RECORD** and **MERGE**, and their counterparts **RECORDREFINED** and **MERGEREFINED**, used for introspective context-sensitivity—as explained below. **RECORD** and **MERGE** follow the usage and naming convention of earlier work [12, 24]. **RECORD** takes all available information at the allocation site of an object and combines it to produce a new heap context, while **MERGE** takes all available information at the call site of a method and combines it to create a new calling context (or just "context"). Heap contexts qualify heap objects in order to provide more fine-grained differentiation than mere allocation sites. Calling contexts are used to qualify method calls, i.e., they are used for local variables in a program. In this way, variables that pertain to different invocations of the same method are distinguished, as much as the context granularity allows. These functions are sufficient for modeling a very large variety of context-sensitive analyses. Explaining the different kinds of context-sensitivity produced by varying **RECORD** and **MERGE** is beyond the scope of this paper—this approach is inherited from past literature[12, 24]. To give a single example, however, a 1-call-site-sensitive analysis with a context-sensitive heap has $C = HC = I$ (i.e., both the context and the heap context are a single instruction), **RECORD** (*heap, ctx*) = ***ctx*** and **MERGE** (*heap, hctx, invo, callerCtx*) = ***invo***. That is, when a method is called, its context is its call site (*invo*) and when an object is allocated, its heap context is the context (*ctx*) of the allocating method (i.e., the call site that invoked the allocating method).

The **RECORDREFINED** and **MERGEREFINED** constructors are directly analogous to **RECORD** and **MERGE** but just apply to different program points. These constructors are the machinery for introspective context-sensitivity: they vary the context-sensitivity of the analysis for a subset of the heap objects and methods.

Analysis logic. The rule syntax in Figure 3 is simple: the left arrow symbol (⟵) separates the inferred facts (i.e., the *head* of the rule) from the previously established facts (i.e., the *body* of the rule). For instance, the first rule states that, if we have computed a call-graph edge between invocation site *invo* and method *meth* (under some contexts), then we infer an inter-procedural assignment to the i-th formal argument of *meth* from the i-th actual argument at *invo*, for every i.

The last rule (in duplicate) is the most involved. It states that if the original program has an instruction making a virtual method call over local variable *base* (this is an input fact), and the computation so far has established that *base* can point to heap object *heap*

[3] REACHABLE is somewhat of a special case, since we assume it is also used as an input relation: it needs to initially hold methods that are always reachable, such as the programs's `main` method, the constructor of class `java.lang.ClassLoader`, and more. We ignore this technicality in the model, rather than burden our rules with a separate input relation.

under a context for which the method is reachable, then the called method is looked up by-signature inside the type of *heap* and several further facts are inferred: that the looked up method is reachable, that it has an edge in the call-graph from the current invocation site, and that its *this* variable can point to *heap*. Additionally, the **MERGE/MERGEREFINED** function is used to possibly create (or look up) the right context for the current invocation.

Note that the rules that use context constructors (**RECORD** and **MERGE**) are essentially duplicated. (In the full implementation, there are some two-dozen rules that construct new contexts, instead of the two in the model, and all are duplicated accordingly.) Each rule has two versions: one for objects (resp. method calls) that should have a default context and one for those that should have a different, refined context. Therefore, we can effect any change we want to the context-sensitivity of an analysis, on a per-object/per-site basis, by supplying the right input relations OBJECTTOREFINE or SITETOREFINE and setting the appropriate constructors, **RECORDREFINED** and **MERGEREFINED** to implement a different flavor of context-sensitivity. We discuss such options next.

3. How To Selectively Refine

The model of the previous section allows configurability of context-sensitivity in a large variety of ways. For instance, some methods (or some call sites) can be analyzed with object-sensitivity while others are analyzed with call-site-sensitivity, of any depth. One aspect to determine, therefore, is the two analyses that will be used in different program points.

Another question is how to populate the OBJECTTOREFINE and SITETOREFINE input relations. One could attempt to do so by mere static inspection of the program at the syntax and type level. For example, methods containing cast statements or methods with weak type information in their signature (e.g., `Object`-typed arguments) can be analyzed with a higher context depth. In our work, we have failed to identify such surface heuristics that would yield benefit. Instead, we rely on running a quick analysis and then querying its results in order to determine which sites can be analyzed more precisely.

Introspective context-sensitivity. Our approach consists of running the analysis *twice*. The first time, OBJECTTOREFINE and SITETOREFINE are empty. The **MERGE/RECORD** context constructors are set so that an inexpensive but scalable analysis is performed. In our experimental setting, these constuctor functions return a unique constant value, $\star$, resulting in a context-insensitive analysis:

RECORD (*heap,ctx*) = $\star$
MERGE (*heap, hctx, invo, ctx*) = $\star$

The **MERGEREFINED** and **RECORDREFINED** constructors are set to implement an expensive context-sensitive analysis, following past techniques [12]. Yet, these constructors are not relevant in the first analysis run, since the rules employing them are predicated on having elements in SITETOREFINE and OBJECTTOREFINE, respectively.

Subsequently, we use the results of the context-insensitive analysis to compute which program elements to refine (i.e., to populate the SITETOREFINE and OBJECTTOREFINE relations), and run the analysis a second time. The result is that a subset of the program elements are analyzed context-sensitively, while the rest are analyzed context-insensitively even during the second analysis run. In practical terms, the former set is larger than the latter: we focus on identifying a relatively small number of program elements that may disproportionately affect analysis costs and to analyze them context-insensitively, while the majority of program elements are analyzed context-sensitively.[4]

It is worth emphasizing again that the two runs of the analysis use identical code—the only difference is that predicates OBJECTTOREFINE and SITETOREFINE are empty in the first run but populated in the second.

Metrics and heuristics. The main challenge is to identify a program query/client analysis (over the results of a context-insensitive points-to analysis) to predict which program elements should not be refined. Our criterion is based on cost rather than expected benefit, since the latter is very hard to estimate in an all-points (as opposed to a client- or demand-driven) program analysis. There are several cost metrics that we can mix-and-match to create introspective analysis heuristics. Examples include:

1. Compute at every invocation site the cumulative size of all points-to sets of actual arguments to the method call. (This is the argument *in-flow* of the method call.)
2. Compute for every method the cumulative (or maximum, for a variant of the metric) size of points-to sets over all local variables. (This is the method's *total points-to volume* or *max var-points-to*.)
3. Compute for each object (i.e., allocation site) the maximum (or total, for a variant of the metric) field points-to set over all of its fields (This is the object's *max field points-to*, or *total field points-to*.)
4. Compute for every method the maximum max field-points-to (metric 3) among objects pointed to by the method's local variables. (This is the method's *max var-field points-to*.)
5. Compute for each object (i.e., allocation site) the number of local variables pointing to it. (This is the object's *pointed-by-vars* metric.)
6. Compute for each object (i.e., allocation site) the number of object-and-field pairs pointing to it. (This is the object's *pointed-by-objs* metric.)

As can be seen, these metrics can vary in sophistication but all of them attempt to estimate the cost that will be incurred if the same method or allocation site were to be analyzed context-sensitively. Indeed, our emphasis is not on the sophistication of the metrics or on their fine-tuning. Instead, it is on their simplicity and ease of composition so that one can create parameterizable analyses: a knob for adjusting the precision/scalability tradeoff. For example, we propose two heuristic combinations of these metrics:

- *Heuristic A*: Refine all allocation sites except those with a *pointed-by-vars* (metric #5) higher than a constant K. Refine all method call sites except those with either an *in-flow* (metric #1) higher than a constant L or a *max var-field points-to* (metric #4) higher than a constant M.
- *Heuristic B*: Refine all method calls sites except those that invoke methods with a *total points-to volume* (metric #2) above a constant P. Refine all object allocations except those for which the product of *total field points-to* and *pointed-by-vars* (metrics #3 and #5) exceeds a constant Q. The product of these two metrics can be seen as an object's total potential for weighing down the analysis.

[4] Since sets SITETOREFINE and OBJECTTOREFINE are much smaller than their complements, it is efficient to compute them in complement form. Our implementation offers this option with extra flags, not captured in the model of Section 2.

These heuristics are themselves tunable, by adjusting the constant parameters. In the rest of the paper, when we refer to *Heuristic A* in measurements, the values of K, L, M will be 100, 100, and 200, respectively; when we refer to *Heuristic B*, the values of P and Q will both be 10000. The point of picking clear-cut reference numbers is to argue that the value of the technique does not come from excessive tuning but from the underlying power of the introspective analysis idea—even relatively large variations of these numbers make scarcely any difference in the total picture of results over multiple programs.

Discussion: Intuition. Our heuristic approach is based on the insight that there are many program elements whose analysis cost is vastly disproportionate to their importance. If such elements are analyzed less precisely, the analysis will avoid significant burden without incurring large precision losses. (Note that our heuristics try to estimate "disproportionate cost" but have no way of estimating the "importance" of a program element. It would be an interesting direction for future work to estimate this importance, i.e., to define metrics that capture the extent of the impact of a program element's precision on all other program elements.)

These heuristics estimate the potential cost of a context-sensitive analysis based on observations on the structure of the analysis computation. First, the cost of an analysis is often determined by the size of the main two relations it computes: VARPOINTSTO and FLDPOINTSTO. In a context-sensitive setting, the size of these relations can grow dramatically. We have no *a priori* knowledge of which program elements will contribute many tuples to the context-sensitive version of these relations, but we have this information for the context-insensitive version of the relations. Thus, most of the metrics (e.g., #1, #2, #3, #5, #6) directly count the number of tuples in which a program element participates, to use as an indication of its cost contribution. Adding more context will potentially just multiply this contribution by the number of distinct contexts.

At the same time, other metrics (e.g., #4, and the product of #3 and #5) are combinatorial: they combine two points-to relationships and see what the cost may be. Such combinations try to capture computation costs not reflected in the size of the final result. To see this, consider that the same context-sensitive points-to fact (i.e., a tuple in the final VARPOINTSTO relation) can be computed in a multitude of ways. Consider, for instance, our rule from Section 2 that derives a var-points-to relationship from a field-points-to one:

```
VARPOINTSTO (to, ctx, heap, hctx) ←
  LOAD (to, base, fld), VARPOINTSTO (base, ctx, baseH, baseHCtx),
  FLDPOINTSTO (baseH, baseHCtx, fld, heap, hctx).
```

All information on the field (*fld*), holder heap object (*baseH*), and its context (*baseHCtx*) is dropped in the final result. The exact same tuple of the VARPOINTSTO relation can be produced for different fields and holder heap objects, resulting in extra computation cost, and this cost will be multiplied for a context-sensitive analysis, when heap context is added. The product of metrics #3 and #5 aims to capture such hidden costs.

Implementation. The above metrics and heuristics can be easily implemented as short analyses over the result of a context-insensitive points-to analysis. For instance, the implementation of the *in-flow* metric (#1) is the following Datalog query, which defines an intermediate predicate and aggregates over it. ("_" is a nameless variable denoting any value, and *agg*<...> denotes an aggregation operation—in this case a total count of matching tuples.)

```
HEAPSPERINVOCATIONPERARG (invo, arg, heap) ←
  CALLGRAPH (invo, _, _, _),
  ACTUALARG (invo, _, arg),
  VARPOINTSTO (arg, _, heap, _).

INFLOW (invo, result) ←
  agg<result = count()>
  (HEAPSPERINVOCATIONPERARG (invo, _, _)).
```

Our full implementation can be found in the Doop framework distribution, as described in Section 7.

4. Evaluation

Our evaluation setting uses the LogicBlox Datalog engine, v.3.9.0, on a Xeon E5530 2.4GHz machine with only one thread running at a time and 24GB of RAM. We analyze the DaCapo benchmark programs (v.2006-10-MR2) with Open JDK 1.6.0_24. We run all benchmarks with default Doop settings, including full reflection support. We selected *a priori* 6 of the Dacapo benchmarks as our experimental subjects. These are the programs that exhibit scalability problems based on past literature [12]: other benchmarks typically run in half the time of the fastest benchmark of our set for deep context-sensitive analyses. Since our technique is explicitly not a "first line of defense", benchmarks that are already certain to scale are out of scope.

We evaluate two variants of introspective context-sensitivity corresponding to *Heuristic A* and *Heuristic B* from Section 3. As discussed earlier, applying our heuristics results in a relatively small number of call sites and objects to avoid refinement. Figure 4 shows these statistics. As can be seen, *Heuristic A* is much more aggressive in preventing refinement, whereas *Heuristic B* is quite selective. In both cases, however, the program elements that are refined are the overwhelming majority.

	Call Sites		Objects	
	Heur. A	Heur. B	Heur. A	Heur. B
bloat	28.0 %	0.7 %	12.3 %	7.1 %
chart	13.7 %	3.4 %	12.0 %	5.0 %
eclipse	14.5 %	0.0 %	11.0 %	5.0 %
hsqldb	30.0 %	1.1 %	16.8 %	14.0 %
jython	36.0 %	3.0 %	25.0 %	18.8 %
pmd	12.5 %	0.0 %	10.5 %	5.3 %
xalan	18.0 %	0.0 %	13.0 %	7.8 %
average	21.81 %	1.18 %	14.37 %	9.0 %

Figure 4. Number of call sites and objects selected to **not** be refined by each introspective variant. All results are rounded to the first decimal digit.

The results of our performance experiments are shown in Figures 5, 6, and 7. We test the three main flavors of context-sensitivity: object-sensitivity [19, 20], call-site-sensitivity [22, 23], and type-sensitivity [24]. The three flavors have very different profiles of practical use and scalability, as detailed next.

Object-sensitivity. Deep-context object-sensitive analyses are the most precise in practice, but do not always scale well. Starting from a 2-object-sensitive analysis with a (1-)context-sensitive heap (2objH), we define our two introspective versions (2objH-IntroA and 2objH-IntroB for *Heuristic A* and *Heuristic B*, resp.). Figure 5 plots first the execution time and then three precision metrics for all analyses. In all cases *lower is better*. There is no real "metric" for precision, since each client may have unique needs, but our three metrics together should yield a reasonable projection of precision. Note that since there is no "ground truth" for the ideal value of precision metrics, their chart scales are arbitrary (and differences are not as visually pronounced as could be because of plotting multiple benchmarks on a single chart) but the insensitive/2objH analyses serve as upper/lower reference markers in practice. We use a 90min timeout. The jython and hsqldb benchmarks did not terminate for 2objH, and jython did not terminate for 2objH-IntroB

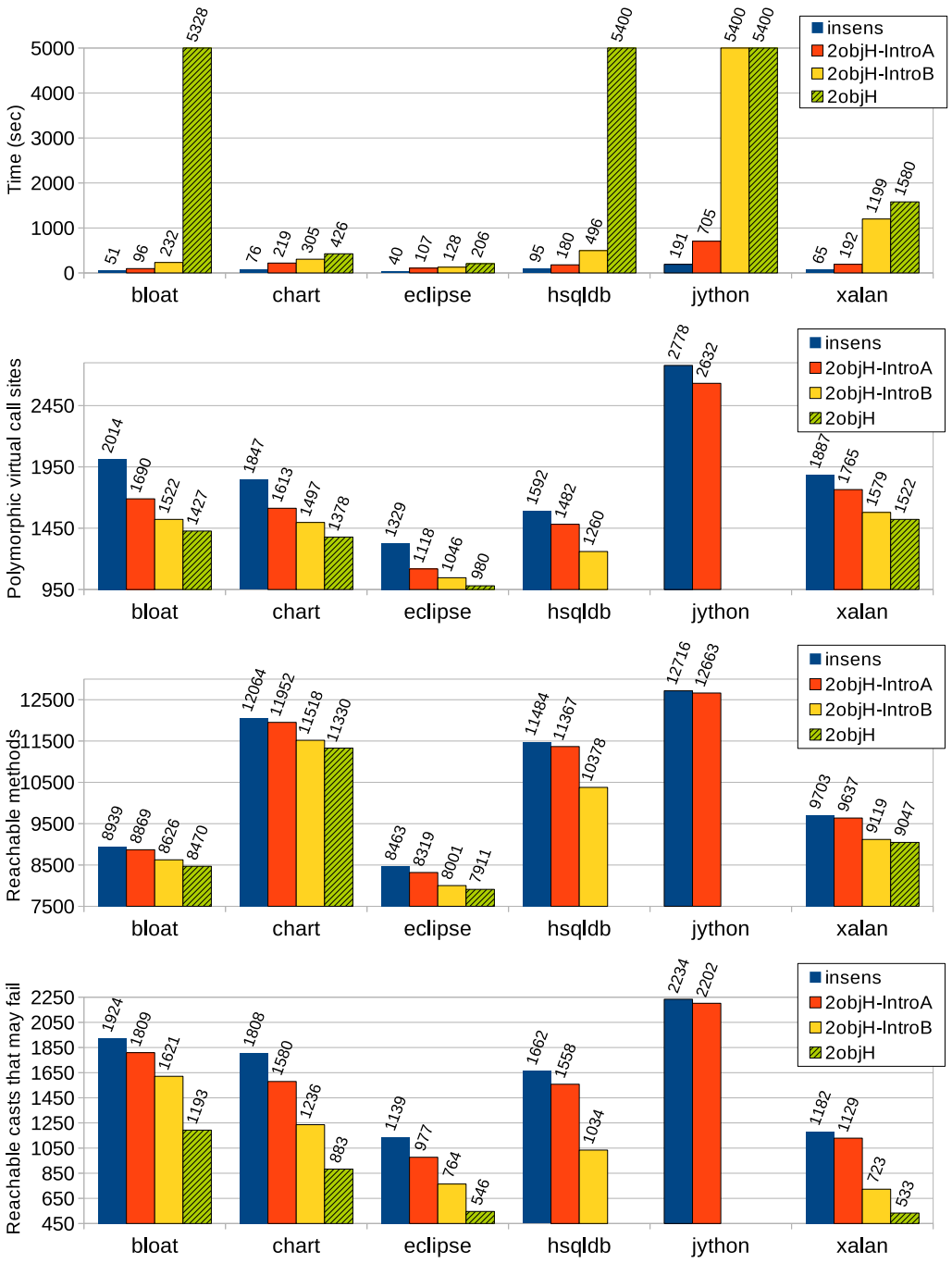

Figure 5. Performance and precision (3 separate metrics: calls that cannot be devirtualized, reachable methods, casts that cannot be eliminated) for introspective context-sensitive variants of a 2objH analysis, compared with baselines (2objH and insensitive).

either. We indicate non-termination with full bars in the top (time) chart and the absence of bars in the bottom three (precision) charts.

As can be seen, the two introspective variants scale much better than the full 2objH analysis. Indeed, IntroA scales to all benchmarks, while showing significant precision gains over an insensitive analysis. IntroB is even more precise: it covers *more than two-thirds* of the precision advantage of 2objH over an insensitive analysis for most benchmarks and precision metrics, while scaling significantly better.

Type-sensitivity. Type-sensitivity is designed with the explicit purpose of providing more scalability than object-sensitivity but in a very different manner: instead of avoiding high context depths, type-sensitivity makes each context element coarser. Thus it is doubly interesting to see if introspection can add benefit to type-sensitive analyses. Type-sensitivity is not immune to the patholo-

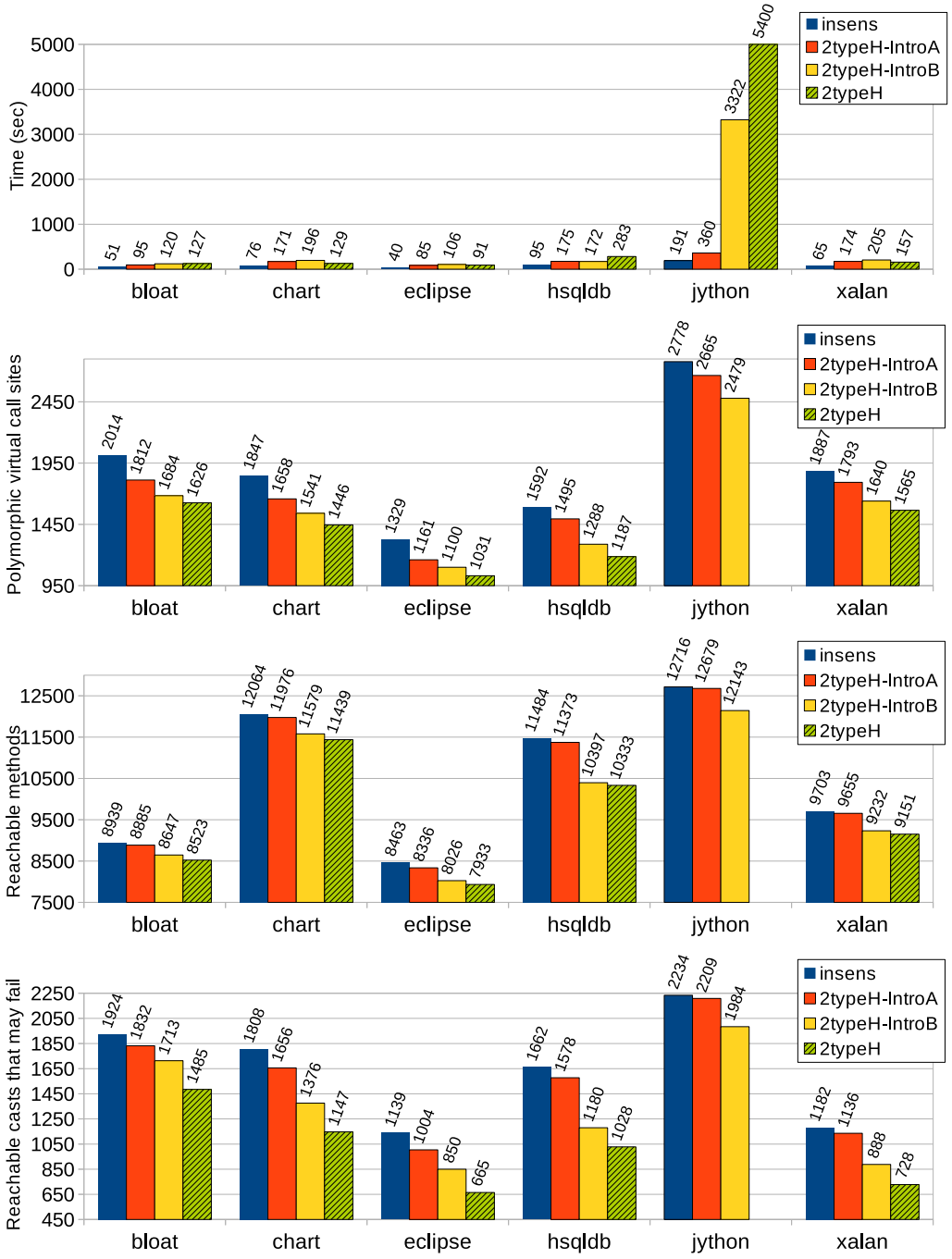

Figure 6. Performance and precision (3 separate metrics: calls that cannot be devirtualized, reachable methods, casts that cannot be eliminated) for introspective context-sensitive variants of a 2typeH analysis, compared with baselines (2typeH and insensitive).

gies of object-sensitivity: for instance, in our benchmark set it does not scale to jython.

Figure 6 shows our results, plotting variants of a 2-type-sensitive analysis with a (1-)context-sensitive heap (2typeH), and following the same conventions as earlier. (The insensitive baseline is inherited and not re-run.) As can be seen, the IntroB version scales to all programs while typically maintaining very good precision—often close to the full 2typeH. The IntroA version has the desirable feature of near-perfect scalability: its maximum runtime for *any* benchmark is 360sec. At the same time it exhibits precision gains compared to a context-insensitive analysis, although these are noticeably lower than the precision gains of IntroB.

Call-site-sensitivity. Call-site-sensitivity is the traditional flavor of context-sensitivity—a virtual synonym for the term. In practice, call-site-sensitivity is quite good for some analysis clients but almost never scalable at context depths greater than 1.

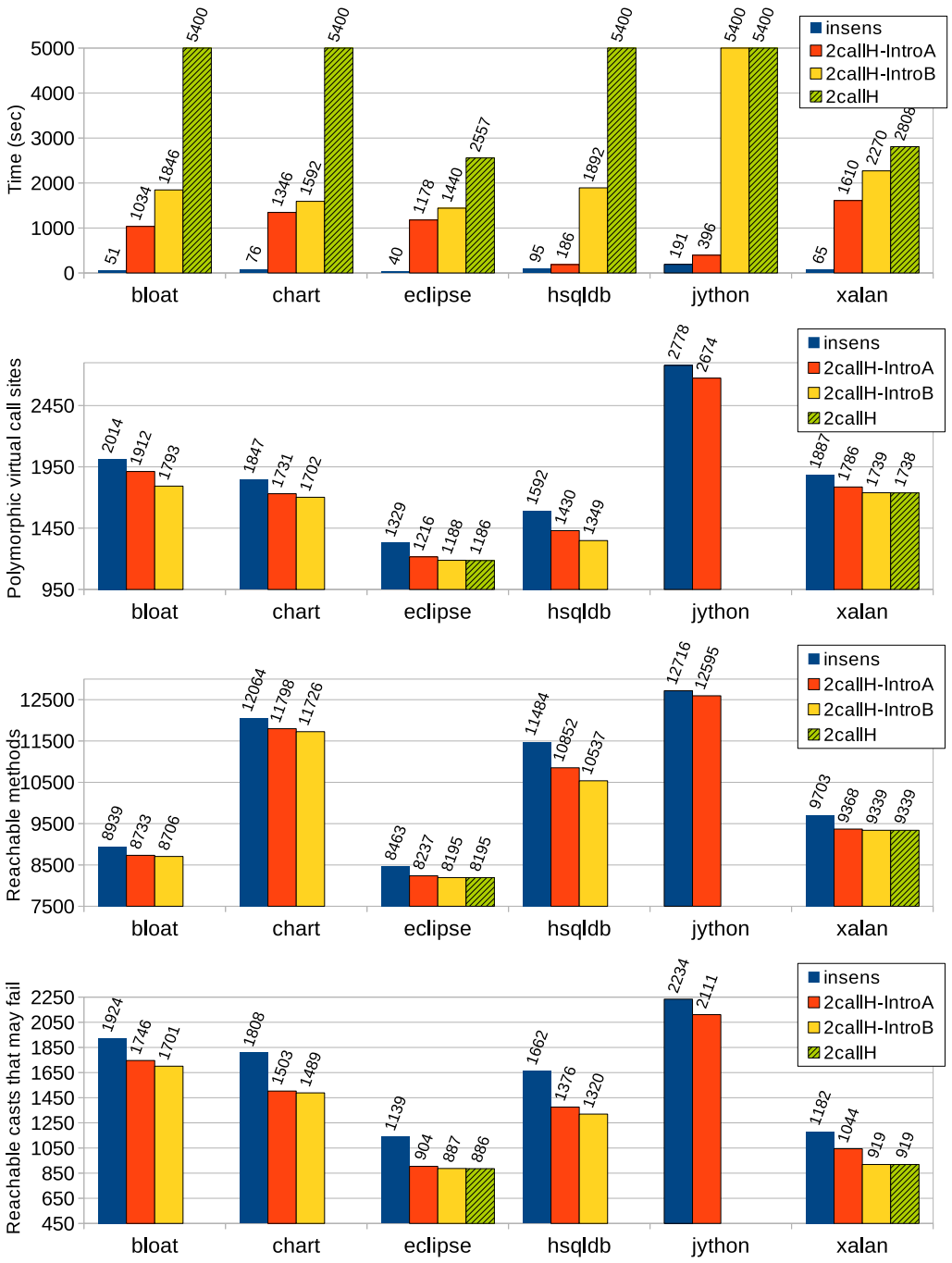

Figure 7. Performance and precision (3 separate metrics: calls that cannot be devirtualized, reachable methods, casts that cannot be eliminated) for introspective context-sensitive variants of a 2callH analysis, compared with baselines (2callH and insensitive).

As Figure 7 shows, introspective context-sensitivity performs remarkably well when applied to a 2-call-site-sensitive analysis with a (1-)context-sensitive heap (2callH). The base 2callH analysis does not terminate for 4-out-of-6 of our benchmarks, while introspective analyses terminate either for all (IntroA) or for nearly all (5-out-of-6 for IntroB). Furthermore, IntroB seems to achieve the full precision of 2callH for the two benchmarks for which the latter yields results, and for all different metrics! Combined with the across-the-board scalability gains shown in the timing chart, this confirms the effectiveness of introspection for tuning out extreme analysis costs. IntroA is not far behind in precision, obtaining more than two-thirds of the precision gains of IntroB for most metrics and benchmarks.

Discussion. The above timings of introspective context-sensitivity do not include the cost of first running a context-insensitive analysis, and other timing overheads (relatively constant

at about 100sec) related to computing the objects and sites to refine and re-running an analysis (our current implementation saves the first-run database and re-generates it from scratch). We did not include these numbers in the timings in order to keep the presentation simpler but also because (a) our emphasis is on scalability and not on small-scale speed gains—we consider small differences in timings, e.g., in the chart and eclipse benchmarks of Figure 5, to be negligible for our purposes; and (b) these constant overheads can be factored out—e.g., with minor engineering we could have incurred them only once per benchmark and not once per run of every introspective analysis variation.

Based on our experimental results, introspective context-sensitivity achieves its goal: it offers a knob for users to select points in the scalability/precision spectrum. The tradeoffs of cost and precision exhibited by *Heuristic A* and *Heuristic B* are illustrative. Not only do these heuristics yield different options (more precision vs. more scalability) but they are also very consistent in their tradeoff, throughout multiple benchmarks and analysis flavors.

Finally note that we used identical introspection heuristics (*Heuristic A* and *Heuristic B*) with the same constants (see Section 3) for all three context-sensitivity flavors and for all benchmarks. This suggests that there are significant opportunities for further tuning: different heuristics can be used, the constants can be optimized, the constants or the heuristics can be adapted per-benchmark or per-context flavor. However, the goal of our experiments is not to squeeze out a few percentage points of speedup but to show that the simple idea of introspective context-sensitivity can easily offer very useful tradeoffs in scalability and precision.

5. Related Work

The effort to tune the context-sensitivity of an analysis is pervasive in the literature. Nevertheless, most approaches fundamentally differ from ours, either by trying to vary context-sensitivity based on syntactic properties or by trying to focus on only a part of the program that matters for answering a given query. In contrast, we attack the context-sensitive scalability problem head-on, in the all-points points-to analysis setting, with context used all over the program and library.

Typical scalable points-to analysis frameworks such as Wala [6] and Doop [2] employ a multitude of low-level heuristics for tuning the precision and scalability of an analysis. These include using extra context for collection classes, using a heap context for arrays in an analysis without a context-sensitive heap, allocating strings or exceptions context-insensitively, treating library factory methods with deeper context, etc. Such heuristics are typically user-selected and prominent in the documentation of the respective frameworks, and have also appeared in the literature (e.g., [11, 27]). However, all such approaches are mere hard-coded heuristics and do not address the major scalability problem that our approach aims to solve. The scalability issues identified in earlier literature and discussed throughout this paper are present after all such heuristics have been employed.

A more general approach is *hybrid* context-sensitivity, which consists of treating virtual and static method calls differently [12]. Such a hybrid analysis attempts to emulate call-site-sensitivity for static method calls and object-sensitivity for dynamic calls. The approach becomes interesting when context is deep (e.g., how are context elements merged when a dynamic call is made inside a static call?). Nevertheless, the hybrid context-sensitivity approach does not change the essence of the problem we are trying to solve. For hard-to-analyze applications, hybrid context-sensitive algorithms are equally unscalable as their component algorithms. For the purposes of our experimental study, which only tests the scalability of heavyweight benchmarks, hybrid context-sensitivity is virtually indistinguishable from object-sensitivity.

More interesting applications of selective context-sensitivity have been explored in the context of *demand-driven* or *client-driven* pointer analysis. Such analyses reduce cost by computing only those results that are necessary for a specific query at a given program location, or by taking into account the precision needs of a specific client [8, 10, 25, 26, 32].

In the demand-driven and client-driven space, refinement-based analyses have been used in work such as that of Guyer and Lin [8], of Sridharan and Bodík [25], and of Liang and Naik [16]. Guyer and Lin [8] automatically adjust the precision of a points-to analysis based on the needs of a client analysis. The adjustments concern flow- and context-sensitivity per-program-point. The analysis maintains a directed dependence graph that connects program points to *polluting assignments*: analysis points where precision may have been lost.

Sridharan and Bodík [25] introduce refinement-based analysis as a way to adaptively increase the precision characteristics of an existing analysis algorithm when a client analysis is not satisfied with the result. The approach allows turning on field-sensitivity, as well as higher call-site-sensitivity for an analysis algorithm. Yet, unlike ours, it is not a general approach that can apply to any kind of context and a large number of different algorithms.

Liang and Naik's "pruning" approach [16] consists of first computing a coarse over-approximation of the points-to information, while keeping the provenance of this derivation, i.e., recording which input facts have affected each part of the output. The input program is then pruned so that parts that did not affect the interesting points of the output are eliminated. Then a highly context-sensitive precise analysis is run, in order to establish the desired property. This approach is similar to introspective context-sensitivity in that the analysis is run twice and a separate query over the first-run result determines the second run's characteristics. Nevertheless, our approach requires no provenance computation (which is unlikely to scale for an all-points analysis) and works even when we want answers for the entire program and any possible client analysis—i.e., when pruning is not possible.

All of the above demand- or client-driven approaches can be viewed as complements of our introspective context-sensitivity. In the client-driven world, it is possible to estimate the *benefit* that a more precise analysis may yield: either the client is happy with the current level of precision (which implies there is no further benefit to be obtained) or it is not, in which case more precision should be added. In our all-points pointer analysis problem we have no such information. This motivates our *cost*-based heuristics, which attempt to estimate "what can go wrong" when more precision gets added, as opposed to "what can be gained", as in client-driven techniques.

6. Conclusions

We introduced introspective context-sensitivity: an approach to making context-sensitive analyses scale. The approach consists of defining an analysis with two separate kinds of context. Each program element is analyzed with one kind, selected based on external input. Then, by first running an inexpensive context-insensitive analysis, we can identify program elements that should be treated with a more precise context and others that should be treated less precisely to avoid an explosion in complexity. Our technique applies to any kind of context abstraction and yields scalability *à la carte*: the user can select a scalability profile and achieve it for a price in precision. As shown in our experiments, this price is not too steep. The precision loss of introspective context-sensitivity can be minuscule (as is for call-site-sensitive analyses), while the scalability gain is substantial.

We believe that introspective context-sensitivity is a big step forward in pointer analysis. It is not just an effective technique, but

an effective technique that addresses the major current pain point in practical applications of points-to analyses.

7. Artifact

Our implementation is integrated in the latest release of the Doop framework, available at `http://doop.program-analysis.org/`. Doop itself is the main and ongoing artifact of our work. For ease of reference, the specific analyses of this paper, with the exact experimental settings described, can also be found separately labeled (`PLDI14`) in the Doop distribution.

Acknowledgments

We thank the anonymous reviewers, as well as the Artifact Evaluation reviewers for their valuable comments. We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant (PADECL) and a European Research Council Starting/Consolidator grant (SPADE); and by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award (MORPH-PL).

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, New York, NY, USA, 2009. ACM.

[3] S. Chong. Personal communication, 2013.

[4] C. Cifuentes. Personal communication, 2013.

[5] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *Int. Conf. on Software engineering (ICSE)*, pages 391–400, New York, NY, USA, 2008. ACM.

[6] S. J. Fink. T.J. Watson libraries for analysis (WALA). `http://wala.sourceforge.net`.

[7] 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*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[8] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, Berlin, Heidelberg, 2003. Springer-Verlag.

[9] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with Datalog. In *European Conf. on Object-Oriented Programming (ECOOP)*, pages 2–27. Spinger, 2006.

[10] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 24–34, New York, NY, USA, 2001. ACM.

[11] G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in Java points-to analysis. In *Compiler Construction*, Mar. 2013.

[12] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Conf. on Programming Language Design and Implementation (PLDI)*. ACM, June 2013.

[13] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Symposium on Principles of Database Systems (PODS)*, pages 1–12, New York, NY, USA, 2005. ACM.

[14] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53, 2008.

[15] O. Lhoták and L. J. Hendren. Context-sensitive points-to analysis: Is it worth it? In A. Mycroft and A. Zeller, editors, *CC*, volume 3923 of *Lecture Notes in Computer Science*, pages 47–64. Springer, 2006.

[16] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 590–601, New York, NY, USA, 2011. ACM.

[17] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of Javascript applications in the presence of frameworks and libraries. Technical Report MSR-TR-2012-66, Microsoft Research, July 2012.

[18] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 305–315. ACM, June 2010.

[19] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–11, New York, NY, USA, 2002. ACM.

[20] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[21] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.

[22] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*, pages 189–233, Englewood Cliffs, NJ, 1981. Prentice-Hall, Inc.

[23] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[24] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity (the making of a precise and scalable pointer analysis). In *Symposium on Principles of Programming Languages (POPL)*, pages 17–30. ACM Press, Jan. 2011.

[25] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 387–400, New York, NY, USA, 2006. ACM.

[26] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 59–76, New York, NY, USA, 2005. ACM.

[27] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 87–97, New York, NY, USA, 2009. ACM.

[28] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Int. Conf. on Compiler Construction (CC)*, pages 18–34, 2000.

[29] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[30] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[31] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 131–144, New York, NY, USA, 2004. ACM.

[32] X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *Symposium on Principles of Programming Languages (POPL)*, pages 197–208, New York, NY, USA, 2008. ACM.