Hybrid Context-Sensitivity for Points-To Analysis

George Kastrinis Yannis Smaragdakis

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

Abstract

Context-sensitive points-to analysis is valuable for achieving high precision with good performance. The standard flavors of context-sensitivity are call-site-sensitivity (kCFA) and object-sensitivity. Combining both flavors of context-sensitivity increases precision but at an infeasibly high cost. We show that a selective combination of call-site- and object-sensitivity for Java points-to analysis is highly profitable. Namely, by keeping a combined context only when analyzing selected language features, we can closely approximate the precision of an analysis that keeps both contexts at all times. In terms of speed, the selective combination of both kinds of context not only vastly outperforms non-selective combinations but is also faster than a mere object-sensitive analysis. This result holds for a large array of analyses (e.g., 1-object-sensitive, 2-object-sensitive with a context-sensitive heap, type-sensitive) establishing a new set of performance/precision sweet spots.

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 a static program analysis that consists of computing all objects (typically identified by allocation site) that a program variable may point to. The area of points-to analysis (and its close relative, *alias analysis*) has been the focus of intense research and is among the most standardized and well-understood of inter-procedural analyses. The emphasis of points-to analysis algorithms is on combining fairly precise modeling of pointer behavior with scalability. The challenge is to pick judicious approximations that will allow satisfactory precision at a reasonable cost. Furthermore, although increasing precision often leads to higher asymptotic complexity, this worst-case behavior is rarely encountered in actual practice. Instead, techniques that are effective at maintaining good precision often also exhibit better average-case performance, since smaller points-to sets lead to less work.

One of the major tools for exploiting sweet spots in the precision/performance tradeoff has been *context-sensitivity*. Context-sensitivity consists of qualifying local variables and objects with context information: the analysis unifies executions that map to the same context value, while separating executions that map to different contexts. This approach tries to counter the loss of precision that naturally results in any static analysis from conflating information from different dynamic program paths. Two main kinds of context-sensitivity have been explored in the literature: *call-site-sensitivity* [22, 23] and *object-sensitivity* [18, 19, 24].

A call-site-sensitive/kCFA analysis uses method call-sites (i.e., labels of instructions that may call the method) as context elements. That is, the analysis separates information on local variables (e.g., method arguments) per call-stack (i.e., sequence of k call-sites) of method invocations that led to the current method call. Similarly, the analysis separates information on heap objects per call-stack of method invocations that led to the object's allocation. For instance, in the code example below, a 1-call-site-sensitive analysis (unlike a *context-insensitive* analysis) will distinguish the two call-sites of method `foo` on lines 7 and 9. This means that the analysis will treat `foo` separately for two cases: that of its formal argument, `o`, pointing to anything `obj1` may point to, and that of `o` pointing to anything `obj2` may point to.

```
1  class C {
2   void foo(Object o) { ... }
3  }
4
5  class Client {
6   void bar(C c1, C c2) { ...
7     c1.foo(obj1);
8     ...
9     c2.foo(obj2);
10  }
11 }
```

In contrast, object-sensitivity uses object allocation sites (i.e., labels of instructions containing a `new` statement) as context elements. (Hence, a better name for "object-sensitivity" might have been "allocation-site sensitivity".) That is, when a method is called on an object, the analysis separates the inferred facts depending on the allocation site of the receiver object (i.e., the object on which the method is called), as well as other allocation sites used as context. Thus, in the above example, a 1-object-sensitive analysis will analyze `foo` separately depending on the allocation sites of the objects that `c1` and `c2` may point to. It is not apparent from this code fragment neither whether `c1` and `c2` may point to different objects, nor to how many objects: the allocation site of the receiver object may be remote and unrelated to the method call itself. Similarly, it is not possible to compare the precision of an object-sensitive and a call-site-sensitive analysis in principle. In this example, it is not even clear whether the object sensitive analysis will examine all calls to `foo` as one case, as two, or as many more, since this

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

depends on the allocation sites of all objects *that the analysis itself computes to flow into* `c1` *and* `c2`.

The question behind our work is whether the two kinds of contexts can be fruitfully combined, since they are quite dissimilar. In order to address this question, we map the design space of *hybrid call-site- and object-sensitive* analyses and describe the combinations that arise. Naive hybrids, such as always maintaining as context *both* a call-site and an allocation site, do not pay off, due to extremely high cost. For instance, keeping one call-site and one allocation site as context yields a very expensive analysis, on average 3.9x slower than a simple 1-object-sensitive analysis.

However, we find that more sophisticated hybrids are highly beneficial. Specifically, we show that we can switch per-language-feature between a combined context and an object-only context. For instance, contexts for static method calls are computed differently from contexts for dynamic method calls. This approach yields analyses with both low cost and high precision. Furthermore, adapting contexts per program feature defines a complex design space and allows even further optimization. Design choices arise, such as, how should the context adapt when a dynamic method call, or an object allocation are made inside a static method?

The end result is analyses that closely track the precision of combined call-site-and-object-sensitivity while incurring none of the cost. In fact, the cost of the resulting analysis is usually less (and occasionally much less) than that of just an object-sensitive analysis, due to increased precision. This effect is shown to apply widely, to several variants of analyses. Accordingly, this outcome establishes new sweet spots for the analyses most relevant for practical applications: 1-object-sensitive, 2-object-sensitive with a 1-context-sensitive heap, and analogous type-sensitive [24] analyses. For all of them, a selective hybrid context is typically both more precise and faster than the original analysis.

In all, our paper makes the following contributions:

- We model the parameter space of context-sensitive points-to analysis in a way that allows both call-site- and object-sensitivity, as well as combinations and switching of context at key points (virtual calls vs. static calls). In this space, we map out choices that produce entirely different flavors of algorithms.
- We introduce the idea of hybrid call-site- and object-sensitivity where the two kinds of context are freely mixed and the mix is adjusted in response to analyzing different program features. The goal is to achieve the precision of keeping *both* kinds of context together, but at the same cost as keeping only one.
- We implement our approach in the DOOP framework by Bravenboer et al. [4] and apply it to a large variety of algorithms with varying context depth.
- We show experimentally, over large Java benchmarks and the Java JDK, that hybrid context-sensitivity works remarkably well. The selective application of a combined context achieves the same effective precision as keeping both contexts at all times, at a fraction of the cost, and is typically faster even than keeping only an object context. For instance, in the practically important case of a 2-object-sensitive analysis with a context-sensitive heap, we get an average speedup of 1.53x *and* a more precise analysis. Similarly, for the simple and popular 1-object-sensitive analysis, we get an average speedup of 1.12x combined with significant increase in precision.

The rest of the paper introduces a model for points-to analysis (Section 2) and shows how instantiating the model yields several well-known analyses. In Section 3 we discuss the many choices for combining object- and call-site-sensitivity, as well as the most promising points in this design space. Our evaluation results follow in Section 4, before describing related work (Section 5).

2. Modeling of Points-To Analysis

We begin with a concise modeling of the relevant points-to analyses and their context choices.

2.1 Background: Parameterizable Model

We model a spectrum of flow-insensitive points-to analyses and joint (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, resulting in a declarative specification: the order of evaluation of rules or examination of clauses cannot affect the final result. The same abstract model applies to a wealth of analyses. We use it to model a context-insensitive Andersen-style [2] analysis, as well as several context-sensitive analyses, both call-site-sensitive and object-sensitive.

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); d) a "virtual method call" instruction that calls the method of the appropriate signature that is defined in the dynamic class of the receiver object; e) a "static method call" instruction that calls a statically known target method. This language models well the Java bytecode representation, but also other high-level intermediate languages. (It does not, however, model languages such as C or C++ that can create pointers through an address-of operator. The techniques used in that space are fairly different—e.g., [8, 9]—although our main hybrid approach is likely to be applicable. Also, even though we model regular object fields and static methods, we omit static fields. Their treatment is a mere engineering complexity, as it does not interact with context choice.) The specification of our points-to analysis as well as the input language are in line with those in past literature [6, 16], although we also integrate elements such as on-the-fly call-graph construction, static calls, 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 [4, 11, 20, 30, 31] and high-level [5, 7]. Indeed, the analysis we show is a faithful model of the implementation in the DOOP framework [4], upon which our work builds. Our specification of the analysis (Figures 1-2) 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[1] currently contains over 600 rules in the common core of all analyses, and several more rules specific to each analysis, as opposed to the 9 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 [17, 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.

[1] Available at http://doop.program-analysis.org/

```
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
ℕ is the set of natural numbers
C is a set of 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(..)
SCALL (meth : M, invo : I, inMeth : M)             # Class.meth(..)

FORMALARG (meth : M, i : ℕ, arg : V)
ACTUALARG (invo : I, i : 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)
------------------------------------------------------------------
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
MERGESTATIC (invo : I, ctx : C) = newCtx : C
```

Figure 1: Our domain, input relations (representing program instructions—with the matching program pattern shown in a comment—and type information), output relations, and constructors of contexts.

Figure 1 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 three constructor functions, responsible for producing new contexts. Figure 2 shows the points-to analysis and call-graph computation. The rule syntax 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—see later), 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.

We explain the contents of both figures in more detail below:

- The input relations correspond to the intermediate language for our analysis. They are logically grouped into relations that represent instructions and relations that represent name-and-type information. 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, VCALL, and SCALL). Similarly, there are relations that encode pertinent symbol table information. Most of these are self-explanatory but some deserve explanation. 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 on 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 on its first argument (a method invocation site) and returns the local variable at the call-site that receives the method call's return value.

```
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).

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).

MERGESTATIC (invo, callerCtx) = calleeCtx,
REACHABLE (toMeth, calleeCtx),
CALLGRAPH (invo, callerCtx, toMeth, calleeCtx) ←
  SCALL (toMeth, invo, inMeth), REACHABLE (inMeth, callerCtx).
```

Figure 2: Datalog rules for the points-to analysis and call-graph construction.

- There are five output or intermediate computed relations (VARPOINTSTO, . . ., REACHABLE). 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. For instance, INTERPROCASSIGN (which encodes all parameter and return value passing) unifies much of the treatment of static and virtual method calls.

- 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 and a single heap 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**, **Merge**, and **MergeStatic**. The first two follow the usage and naming convention of Smaragdakis et al. [24], while **MergeStatic** is new and used to differentiate the treatment of static calls—this is a crucial element of our approach.

  Record is the function that creates a new heap context. It is invoked whenever an object allocation site (input relation Alloc) is analyzed. Thus, **Record** is only used in the rule treating allocation instructions (3rd rule in Figure 2). **Record** takes all available information at the allocation site of an object and combines it to produce a new heap context. The rule merely says that an allocation instruction in a reachable method leads us to infer a points-to fact between the allocated object and the variable it is directly assigned to.

  Merge and **MergeStatic** are used to create new calling contexts (or just "contexts"). These contexts are used to qualify method calls, i.e., they are applied to all local variables in a program. The **Merge** and **MergeStatic** functions take all available information at the call-site of a method (virtual or static) and combine it to create a new context (if one for the same combination of parameters does not already exist). These functions are sufficient for modeling a very large variety of context-sensitive analyses, as we show in Sections 2.2 and 3.

  Note that the use of constructors, such as **Record**, **Merge**, and **MergeStatic**, is not part of regular Datalog and can result in infinite structures (e.g., one can express unbounded call-site sensitivity) if care is not taken. All our later definitions statically guarantee to create contexts of a pre-set depth.

- The rules of Figure 2 show how each input instruction leads to the inference of facts for the five output or intermediate relations. The most complex rule is the second-to-last, which handles virtual method calls (input relation VCall). The rule says that if a reachable method of the program has an instruction making a virtual method call over local variable *base* (this is an input fact), and the analysis so far has established that *base* can point to heap object *heap*, then the called method is looked up 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** function is used to possibly create (or look up) the right context for the current invocation.

2.2 Instantiating the Model: Standard Analyses

By modifying the definitions of the **Record**, **Merge** and **MergeStatic** functions as well as domains HC and C, one can create endless variations of points-to analyses. We next discuss the most interesting combinations from past literature, before we introduce our own (in Section 3). For every analysis name we also list a common abbreviation, which we often use later.

Context-insensitive (insens). As already mentioned, our context-sensitive analysis framework can yield a context-insensitive analysis by merely picking singleton C and HC sets (i.e., $C = HC = \{\star\}$, where $\star$ is merely a name for a distinguished element) and constructor functions that return the single element of the set:

Record (*heap,ctx*) = $\star$
Merge (*heap, hctx, invo, ctx*) = $\star$
MergeStatic (*invo, ctx*) = $\star$

Note that the absence of contexts does not mean that the identity of input elements is forgotten. Objects are still represented by their allocation site (i.e., the exact program instruction that allocated the object) and local variables are still distinguished (e.g., by their declaration location in the input program). The absence of context just means that there is no *extra* distinguishing information. This can also be seen in the rules of Figure 2, where the *var* and *heap* predicate arguments are present, separately from the context arguments.

1-call-site-sensitive (1call). A 1-call-site-sensitive analysis has no heap context to qualify heap abstractions ($HC = \{\star\}$) and uses the current invocation site as a context ($C = I$). The following definitions describe such an analysis.

Record (*heap, ctx*) = $\star$
Merge (*heap, hctx, invo, ctx*) = ***invo***
MergeStatic (*invo, ctx*) = ***invo***

In words: the analysis stores no context when an object is created (**Record**) and keeps the invocation site as context in both virtual and static calls.

1-call-site-sensitive with a context-sensitive heap (1call+H). The analysis is defined similarly to 1call.[2] The heap context as well as the main context consist of an invocation site ($HC = C = I$).

Record (*heap, ctx*) = ***ctx***
Merge (*heap, hctx, invo, ctx*) = ***invo***
MergeStatic (*invo, ctx*) = ***invo***

In words: the analysis uses the current method's context as a heap context for objects allocated inside the method. The invocation site of a method call is the context of the method for both virtual and static calls.

1-object-sensitive (1obj). Object sensitivity uses allocation sites as context components. A 1-object-sensitive analysis has no heap context ($HC = \{\star\}$) and uses the allocation site of the receiver object as context ($C = H$). The following definitions complete the description.

Record (*heap, ctx*) = $\star$
Merge (*heap, hctx, invo, ctx*) = ***heap***
MergeStatic (*invo, ctx*) = ***ctx***

In words: the analysis stores no context for allocated objects. For virtual method calls, the context is the allocation site of the receiver object. For static method calls, the context for the called method is that of the calling method.

The above definition offers a first glimpse of the possibilities that we explore in this paper, and can serve as motivation. In static calls, the context of the caller method is copied, i.e., the receiver object of the caller method is used as the new context. Why not try **MergeStatic** (*invo, ctx*) = ***invo***, instead of the current **MergeStatic** (*invo, ctx*) = ***ctx***? Isn't it perhaps better to use call-sites to differentiate static invocations, instead of blindly copying the context of the last non-static method called? A simple answer is that *invo* is an entity of the wrong type, since $C = H$. The only entity of type H we have available at a static call-site is the current context, *ctx*. But if we let $C = H \cup I$, we have a context type that is a hybrid of both an allocation site and an invocation site, and which allows the above alternative definition of **MergeStatic**. We explore this and other such directions in depth in Section 3.

2-object-sensitive with a 1-context-sensitive heap (2obj+H). In this case, the heap context consists of one allocation site ($HC = H$)

[2] The standard convention in the points-to analysis literature is to name an analysis first according to the context of methods, and, if a heap context exists, designate it in a suffix such as *context-sensitive heap* or *heap cloning*.

and the context consists of two allocation sites ($C = H \times H$). The definitions of constructor functions are:[3]

RECORD (*heap, ctx*) = ***first(ctx)***
MERGE (*heap, hctx, invo, ctx*) = ***pair(heap, hctx)***
MERGESTATIC (*invo, ctx*) = ***ctx***

In words: the context of a virtual method (see **MERGE**) is a 2-element list consisting of the receiver object and its (heap) context. The heap context of an object (fixed at allocation, via **RECORD**) is the first context element of the allocating method, i.e., the receiver object on which it was invoked. Therefore, the context of a virtual method is the receiver object together with the "parent" receiver object (the receiver object of the method that allocated the receiver object of the virtual call). Again, static calls just copy the context of the caller method.

Although there can be other definitions of the **MERGE** function, yielding alternative 2-obj+H analyses, it has been shown [24] that the above is the most precise and scalable. In intuitive terms, we use as method context the most precise abstraction of the receiver object available to the analysis.

2-type-sensitive with a 1-context-sensitive heap (2type+H). A type-sensitive analysis is step-by-step analogous to an object-sensitive one, but instead of using allocation sites (i.e., instructions) a type-sensitive analysis uses the name of the class containing the allocation site. In this way, all allocation sites in methods declared in the same class (though not inherited methods) are merged. This approximation was introduced by Smaragdakis et al. [24] and yields much more scalable analyses at the expense of moderate precision loss (as we also determine in our experiments).

In order to define type-sensitive analyses we need an auxiliary function which maps each heap abstraction to the class containing the allocation.

$\mathcal{C}_\mathcal{A} : H \to T$

Now we can define a 2type+H analysis by mapping $\mathcal{C}_\mathcal{A}$ over the context of a 2obj+H analysis. The heap context uses a type instead of an allocation site ($HC = T$) and the calling context uses two types ($C = T \times T$).

RECORD (*heap, ctx*) = ***first(ctx)***
MERGE (*heap, hctx, invo, ctx*) = ***pair($\mathcal{C}_\mathcal{A}$(heap), hctx)***
MERGESTATIC (*invo, ctx*) = ***ctx***

Other Analyses. The above discussion omits several analyses in the literature, in order to focus on a manageable set with practical relevance. We do not discuss a 1-object-sensitive analysis with a context-sensitive heap (1obj+H) because it is a strictly inferior choice to other analyses (especially 2type+H) in practice: it is both much less precise and much slower. We do not present other varieties of type-sensitivity for a similar reason. Deeper contexts or heap contexts (e.g., 2call+H, 2obj+2H, 3obj, etc.) quickly make an analysis intractable for a substantial portion of realistic programs and modern JDKs. In short, we focus on the specific analyses (1call, 1call+H, 1obj, 2obj+H, 2type+H) that are of most practical interest: they are quite scalable over a variety of medium-to-large programs, and no other analysis supplants them by being uniformly better in both precision and performance.

3. Hybrid Context-Sensitive Analyses

We can now explore interesting combinations of call-site- and object-sensitivity. The design space is large and we will be selective in our presentation and later experiments. Our choice of analyses in this space leverages insights from past studies on what kinds of context are beneficial.[4] Such insights include:

- A call-site-sensitive heap is far less attractive than an object-sensitive heap. Generally, adding a heap context to a call-site-sensitive analysis increases precision very slightly, compared to the overwhelming cost.
- When there is a choice between keeping an object-context or a call-site-context, the former is typically more profitable. This is well validated in extensive past measurements by Lhoták and Hendren [14], comparing call-site-sensitive and object-sensitive analyses of various depths. In other words, call-site-sensitivity is best added as *extra* context over an object-sensitive analysis and will almost never pay off as a replacement context, for an object-oriented language.

3.1 Uniform Hybrid Analyses

The first kind of context combination is a straightforward one: both kinds of context are kept. We term such combinations *uniform hybrid analyses*. In the variants we describe, a uniform hybrid analysis is guaranteed to be more precise[5] than the base analysis being enhanced. The question is whether such precision will justify the added cost.

Uniform 1-object-sensitive hybrid (U-1obj). Enhancing a 1-object-sensitive analysis with call-site sensitivity results in an analysis with an empty heap context ($HC = \{\star\}$) but with a context that consists of both the allocation site of the receiver object and the invocation site of the method ($C = H \times I$). The following definitions describe the analysis:

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

In words: a virtual method has as context the abstraction of its receiver object, extended with the method's invocation site. A static method keeps a context consisting of the most significant part of the caller's context and the method's invocation site.

Note that under the above definitions, the context of a U-1obj analysis is always a superset of that of 1obj, hence the analysis is strictly more precise.

Uniform 2-object-sensitive with 1-context-sensitive heap hybrid (U-2obj+H). A 2-object-sensitive analysis with a context-sensitive heap can be enhanced in the same way. A heap context consists of an allocation site ($HC = H$) and a method context consists of two allocation sites and one invocation site ($C = H \times H \times I$). The constructor definitions for the analysis are:

RECORD (*heap, ctx*) = ***first(ctx)***
MERGE (*heap, hctx, invo, ctx*) = ***triple(heap, hctx, invo)***
MERGESTATIC (*invo, ctx*) = ***triple(first(ctx), second(ctx), invo)***

In words: an object's heap context is the receiver object of the method doing the allocation. A virtual method's context is its receiver object's allocation site and context (the latter being the allocation site of the object that allocated the receiver), followed by the

[3] We use auxiliary constructor functions *pair*, *triple* and accessors *first*, *second*, and *third*, with the expected meaning, in order to construct and deconstruct contexts with 2 or 3 elements. This has the added advantage that our context-depth is statically bounded—we never create lists of unknown length. Since our most complex constructor is *triple*, the possible number of distinct contexts is cubic in the size of the input program.

[4] We have validated these insights with extensive measurements on our experimental setup, and have generally explored a much larger portion of the design space than is possible to present in our evaluation section.

[5] We use the term "more precise" colloquially. Strictly speaking, the analysis is guaranteed to be "at least as precise".

invocation site of the method. On a static call, the heap part (i.e., first two elements) of the method context is kept unchanged, and extended with the invocation site of the call.

This analysis is also strictly more precise than the "plain" analysis it is based on, 2obj+H. Note that this is achieved partly by placing the receiver object's allocation site in the most significant position of the context triple. In this way, the **Record** function produces the same heap context as 2obj+H on an object's allocation. Alternative definitions are possible for the same sets of contexts, C and HC. For instance, one could choose to place *hctx* in the most significant position. Similarly, one could produce a hybrid analysis based on 2obj+H but with a different kind of heap context, e.g., $HC = I$, therefore using the invocation site in a method's context as an allocation context. These definitions make decisively less sense, however, per the insights mentioned earlier: invocation sites are rarely advantageous as heap contexts, and, similarly, it is not reasonable to invert the natural significance order of *heap* vs. *hctx*. (We have also verified experimentally that such combinations yield bad analyses.)

Uniform 2-type-sensitive with 1-context-sensitive heap hybrid (U-2type+H). Isomorphically to object-sensitivity, we can enhance type-sensitive analyses with call-site information in the same way. When applied to a 2-type-sensitive analysis with a context-sensitive heap, this results in an analysis with a heap context of one type ($HC = T$) and a context of two types and an invocation site ($C = T \times T \times I$)—mirroring the 2-object-sensitive analysis with a context sensitive heap. The definitions are almost identical:

Record (*heap, ctx*) = ***first(ctx)***
Merge (*heap, hctx, invo, ctx*) = ***triple($\mathcal{C}_\mathcal{A}$(heap), hctx, invo)***
MergeStatic (*invo, ctx*) = ***triple(first(ctx), second(ctx), invo)***

3.2 Selective Hybrid Analyses

Another approach to hybrid call-site- and object-sensitive analyses is to maintain a context that varies inside the same analysis. We call such analyses *selective hybrid analyses*, as opposed to the earlier "uniform hybrid" ones. In a selective hybrid analysis, the sets of contexts, C and HC, will be formed as the cartesian product of unions of sets. Depending on the information available at different analysis points where new contexts are formed, we shall create contexts of a different kind, instead of always keeping a combination of rigid form. We have already hinted at such opportunities in Section 2.2: at a static method call, an object-sensitive analysis does not have a heap object available to create a new context, hence it can at best propagate the context of the caller. Yet, an invocation site is available and can be used to distinguish different static calls, as long as we are allowed to use it as context. This observation generalizes: static invocations are a language feature that benefits highly from the presence of call-site-sensitive elements in the context. This is not hard to explain: For object-sensitive analyses, when analyzing a static invocation, we do not have much information to use in creating a new context, in contrast to a "normal" virtual invocation. Consequently, it is beneficial to be able to use the invocation site as a differentiator of static calls.

Selective hybrid analyses are among the most interesting parts of our work and, to our knowledge, have never before arisen in the literature, far less specified, implemented, and evaluated.

Selective 1-object-sensitive hybrid A (S_A-1obj). Trying to selectively enhance a 1-object-sensitive analysis ($HC = \{\star\}$) with call-site sensitive elements, we are presented with two options, relative to how contexts are created in static invocations. The first option is quite simple: we can keep only a single context element in both virtual and static invocations. Consequently, in virtual invocations the context will be an allocation site, but in static invocations it will be an invocation site ($C = H \cup I$). The definitions needed are the following:

Record (*heap, ctx*) = $\star$
Merge (*heap, hctx, invo, ctx*) = ***heap***
MergeStatic (*invo, ctx*) = ***invo***

Note that this analysis is *not* guaranteed to be more precise than the 1obj analysis it is based on. Nevertheless, it should be an excellent reference point for comparison and insights: it will suggest how much precision can be gained or lost by call-site-sensitivity as a replacement of object-sensitivity in static method calls.

Selective 1-object-sensitive hybrid B (S_B-1obj). The second option for a selective hybrid enhancement of a 1-object-sensitive analysis is to add *extra* information to the context of static calls. This means that context in virtual invocations is still an allocation site, but context in static invocations now consists of both the allocation site copied from the caller *and* the invocation site. In this way, $C = H \times (I \cup \{\star\})$. That is, the context can be either just an allocation site or an allocation site and an invocation site. (This could also be written equivalently as $C = H \cup (H \times I)$, but the earlier form streamlines the definitions of constructors, as it makes all contexts be pairs, thus avoiding case-based definitions.) In this way the constructor definitions become:

Record (*heap, ctx*) = $\star$
Merge (*heap, hctx, invo, ctx*) = ***pair(heap, $\star$)***
MergeStatic (*invo, ctx*) = ***pair(first(ctx), invo)***

This analysis has a context that is always a superset of the 1obj context and, therefore, is guaranteed to be more precise.

Selective 2-object-sensitive with 1-context-sensitive heap hybrid (S-2obj+H). When dealing with deeper analyses, the possible design decisions start to vary. For example, for a 2-object-sensitive analysis with a context-sensitive heap, an interesting choice is to have allocation sites as heap contexts ($HC = H$), and for method contexts to keep standard object-sensitive information for virtual calls but favor call-site-sensitivity for static calls. The constructor definitions for the above analysis are:

Record (*heap, ctx*) = ***first(ctx)***
Merge (*heap, hctx, invo, ctx*) = ***triple(heap, hctx, $\star$)***
MergeStatic (*invo, ctx*) = ***triple(first(ctx), invo, second(ctx))***

(In this way, we have $C = H \times (H \cup I) \times (H \cup I \cup \{\star\})$.) Note the interesting behavior of such an analysis: for virtual calls, the context is equivalent to that of 2obj+H. For the first static call (i.e., from inside a virtually called method), the context is a superset of 2obj+H, augmented by an invocation site. For further static calls (i.e., static calls inside statically called methods), however, the analysis favors call-site sensitivity (both the last two elements of context are invocation sites) and otherwise only remembers the most-significant element of the object-sensitive context. (The latter is important for creating high-quality heap contexts, when allocating objects.) It is interesting to see how this analysis fares relative to 2obj+H, since the analyses are in principle incomparable in precision.

Selective 2-type-sensitive with 1-context-sensitive heap hybrid (S-2type+H). Finally, type-sensitive analyses can be enhanced with call-site sensitive information in much the same way. Mirroring our choices in S-2obj+H, the S-2type+H analysis has heap context $HC = T$ and method context $C = T \times (T \cup I) \times (T \cup I \cup \{\star\})$. The constructor definitions are isomorphic to the S-2obj+H analysis:

Record (*heap, ctx*) = ***first(ctx)***
Merge (*heap, hctx, invo, ctx*) = ***triple($\mathcal{C}_\mathcal{A}$(heap), hctx, $\star$)***
MergeStatic (*invo, ctx*) = ***triple(first(ctx), invo, second(ctx))***

Other analyses. The above discussion does not nearly exhaust the space of hybrid combinations. Consider selective hybrids for a 2obj+H analysis: Many more design choices are possible than the one shown. One could change the heap context into an invocation site, or into a union of invocation and call-site ($HC = H \cup I$). This combination is a bad choice, due to the poor payoff of call-site heap contexts. One could create context structures that let call-site- and object-sensitive context elements freely merge, e.g., $C = (H \cup I) \times (H \cup I) \times (H \cup I \cup \{\star\})$. This allows several different definitions of context constructors, but has the drawback of diverging significantly from object-sensitivity (i.e., allowing to skip even the most-significant object-sensitive context element), which misses the well documented precision and performance advantages of object-sensitivity, especially as a heap context.

4. Evaluation

We implemented and evaluated all aforementioned analyses using the DOOP framework [4]. There are interesting and subtle aspects in our measurements, but the executive summary is clear: uniform hybrid analyses are typically not good choices in practice: their precision is offset by a very high performance cost. (A relative exception is the uniform type-sensitive hybrid analysis, U-2type+H, which, although higher-cost, is not prohibitively expensive and offers a reasonable precision/performance tradeoff.) Selective hybrid analyses, on the other hand, are not just interesting tradeoffs but clear winners: they match or (usually) outperform the object-sensitive analyses they are based on, while offering better precision, closely approaching the precision of the much more costly uniform hybrids. Overall, the best analyses in our evaluation set, both for highest-precision and for high performance with good precision, are selective hybrids.

Our evaluation setting uses the LogicBlox Datalog engine, v.3.9.0, on a Xeon X5650 2.67GHz machine with only one thread running at a time and 24GB of RAM (i.e., ample for the analyses studied). We analyze the DaCapo benchmark programs (v.2006-10-MR2) under JDK 1.6.0_37. (This is a much larger set of libraries than earlier work [4, 24], which also results in differences in measurements, since the numbers shown integrate application- and library-level metrics.) (All numbers shown are medians of three runs.) All runtime numbers are medians of three runs. As in other published work [1, 24], jython and hsqldb are analyzed with reflection disabled and hsqldb has its entry point set manually in a special harness.

4.1 Illustration

For an illustration of the precision and performance spectrum, consider Figure 3, which plots analyses on precision/performance axes. The figure plots execution time against precision in the "may-fail casts" metric, i.e., the number of casts that the analysis cannot statically prove safe. Lower numbers are better on both axes, thus an analysis that is to the left and below another is better in both precision and performance. Values that are disproportionately high on the Y axis (i.e., large execution times) are clipped and plotted at the top of the figure, with the actual number included in parentheses. (Note that the Y axis starts at zero, while the X axis starts at an arbitrary point—we cannot know what is the "ideal" reference value for this metric.)

In terms of pre-existing analyses, Figure 3 illustrates what has been past experience: 2obj+H is the most precise analysis, but often heavy. 1obj and 2type+H are both quite fast, with 2type+H also showing very good precision, often approaching 2obj+H. The two call-site-sensitive analyses (1call, 1call+H) are mostly shown for reference and to demonstrate the insights discussed in Section 3. 1call is a fast analysis but vastly imprecise, while 1call+H is a bad tradeoff: its cost grows quite significantly relative to 1call without much precision added—call-site sensitivity is a bad choice for heap contexts.

As can be seen, the selective hybrid analyses (S_A-1obj, S_A-1obj, S-2obj+H, S-2type+H) usually offer an advantage over the corresponding base analysis (1obj, 2type+H, 2obj+H) in both precision and performance. In fact, selective hybrids are typically imperceptibly less precise than the corresponding uniform hybrid, yet much more precise than the base analysis. For instance, the plot points for S-2obj+H are always barely to the right of those for the theoretically more precise U-2obj+H (but significantly lower—uniform hybrids are very expensive), while they are clearly to the left of 2obj+H.

4.2 Detailed Results

Detailed results of our experiments are presented in Table 1. The table shows precision and performance metrics for all analyses. The precision metrics are the average points-to set size (i.e., average over all variables of their points-to sets sizes), the number of edges in the computed call-graph (which is typically a good proxy for the overall precision of the analysis, in broad strokes), and the results of two client analyses: the number of virtual calls that could not be de-virtualized, and the number of casts that could not be statically proven safe. A combination of these four metrics gives a reliable picture of the precision of an analysis. (Note that the average points-to set size alone is not necessarily reliable, because it is influenced by a small number of library variables with enormous points-to sets. For comparison, the *median* points-to set size is 1, for all analyses and benchmarks.)

Performance is shown with two metrics: time and total size of all *context-sensitive* points-to sets. Although time is the ultimate performance metric, it is brittle: one can argue that our time measurements are influenced by a multitude of implementation or environment factors, among which are the choice of underlying data structures, indexing, and the overall implementation of the points-to analysis, especially since it is running on a Datalog engine, with its own complex implementation choices hidden. The context-sensitive points-to set size metric does not suffer from any such measurement or implementation bias. It is the foremost internal complexity metric of a points-to analysis, and typically correlates well with time, for analyses of the same flavor. Note that analysis implementations that fundamentally differ from ours also try hard to minimize this metric in order to achieve peak performance: Lhoták's PADDLE framework [12] is using binary decision diagrams (BDDs) for representing relations. The best BDD variable ordering (yielding "impressive results" [3]) is one that minimizes the total size of context-sensitive points-to sets. In short, it is reasonable to expect that improvements in this internal metric reinforce the verdict of which analysis yields better performance, not just in our setting but generally. Furthermore, the size of context-sensitive points-to sets serves as a valuable indicator of the internally different computation performed by various analyses: Analyses with almost identical precision metrics (e.g., context-insensitive points-to set sizes, call-graph edges) have vastly different context-sensitive points-to set sizes.

Since Table 1 has a high information density, we guide the reader through some of the most important findings below (see also a partial illustration in Figure 3, later).

- ***General observations.*** The analyses shown are in 4 groups of closely related analyses: call-site sensitive, 1-object-sensitive, 2-object-sensitive with a 1-context-sensitive heap, and 2-type-sensitive with a 1-context-sensitive heap. These analyses span a large performance and precision spectrum. For instance, for the chart benchmark, the least precise analysis, 1call, runs for under 5mins and computes an average points-to size of over 45, while the most precise, U-2obj+H, runs for over 53mins and computes

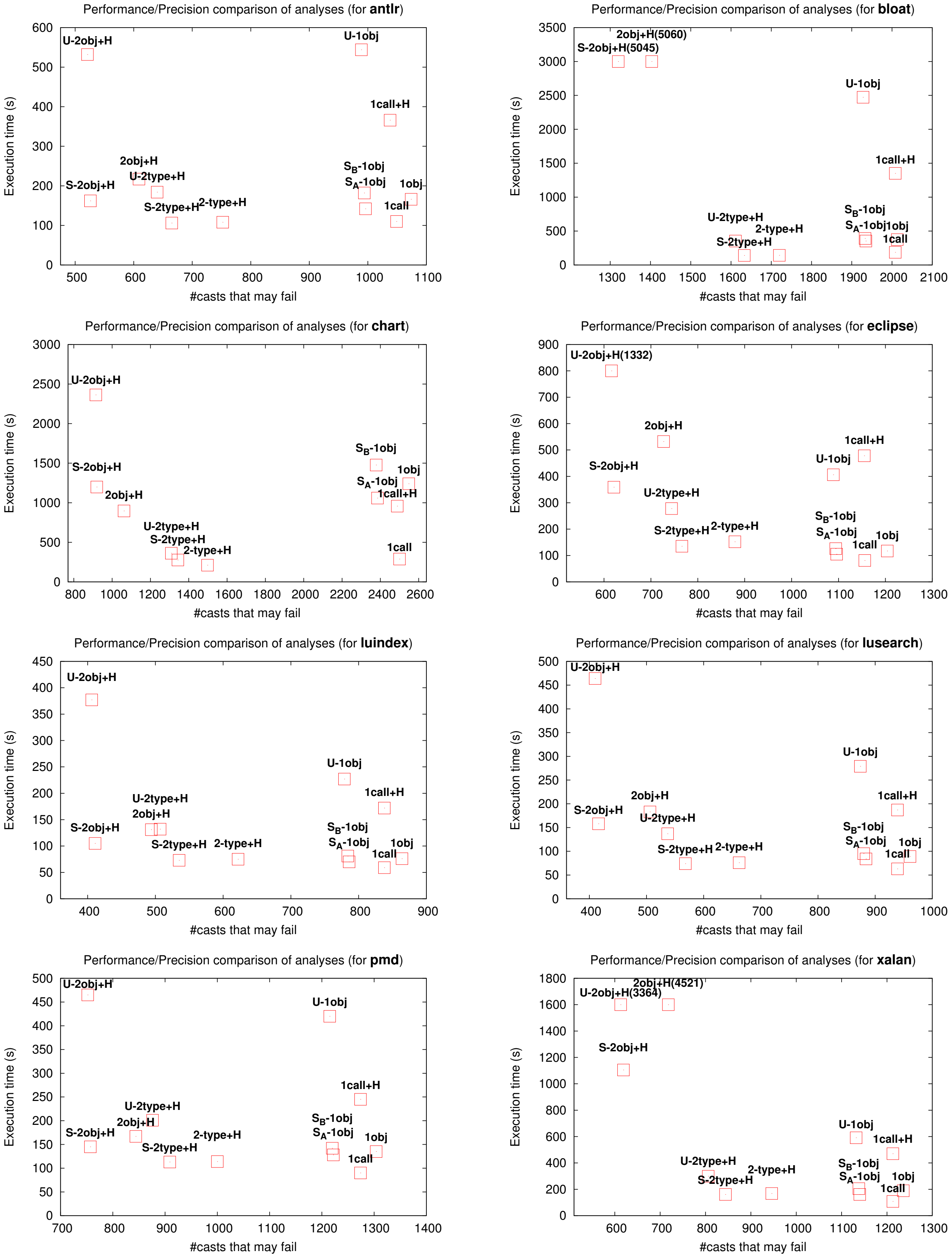

Figure 3: Graphical depiction of performance vs. precision metrics for eight of our benchmarks over all analyses. Lower is better on both axes. The Y axis is truncated for readability. Out-of-bounds points are included at lower Y values, with their real running time in parentheses.

		1call	*1call+H*	*1obj*	*U-1obj*	*S_A-1obj*	*S_B-1obj*	*2obj+H*	*U-2obj+H*	*S-2obj+H*	*2type+H*	*U-2type+H*	*S-2type+H*
antlr	avg. objs per var	29.79	29.58	24.86	24.55	24.90	24.61	6.42	6.26	6.27	17.60	7.14	17.39
	edges (over ∼8.8K meths)	60999	60999	60194	60194	60202	60194	55548	55548	55548	55850	55765	55850
	poly v-calls (of ∼33K)	1994	1994	1933	1933	1936	1933	1707	1707	1707	1759	1746	1759
	may-fail casts (of ∼1.7K)	1049	1038	1074	989	996	994	609	521	526	752	640	665
	elapsed time (s)	110	366	166	544	**142**	182	217	532	**162**	108	184	**106**
	sensitive var-points-to (M)	16	54.8	14.3	65	**10.5**	17.3	19.9	39.5	**13.9**	5.3	8.9	**4.8**
bloat	avg. objs per var	43.96	43.94	41.26	41.16	41.32	41.18	13.25	-	13.10	16.28	14.71	16.10
	edges (over ∼10.2K meths)	70506	70506	69501	69501	69511	69501	60726	-	60726	62115	61753	62115
	poly v-calls (of ∼31K)	2138	2138	2076	2076	2080	2076	1640	-	1640	1888	1827	1888
	may-fail casts (of ∼2.8K)	2008	2008	2013	1928	1935	1933	1403	-	1320	1720	1611	1633
	elapsed time (s)	186	1351	374	2473	**353**	391	5060	-	**5045**	142	353	**140**
	sensitive var-points-to (M)	32.9	150.5	21.9	287.1	**20.1**	24	153.5	-	**149.8**	11.4	30.3	**11**
chart	avg. objs per var	45.12	45.11	40.80	-	40.72	40.11	5.30	4.99	5.00	7.02	5.89	6.57
	edges (over ∼15K meths)	82156	82078	81423	-	81075	81012	59162	59142	59152	62290	62172	62280
	poly v-calls (of ∼35K)	2900	2897	2821	-	2815	2808	1610	1603	1610	1775	1756	1775
	may-fail casts (of ∼3.5K)	2500	2488	2548	-	2385	2378	1062	915	920	1498	1309	1343
	elapsed time (s)	288	957	1240	-	**1059**	1477	**896**	2363	1199	**211**	362	276
	sensitive var-points-to (M)	49.6	120.9	62.5	-	**39.7**	89.7	67.6	115.7	**53**	**13.3**	21.3	16.5
eclipse	avg. objs per var	21.84	21.65	18.65	18.41	18.59	18.43	5.75	5.60	5.61	7.93	6.41	7.61
	edges (over ∼9.3K meths)	53006	53001	52114	51935	51958	51936	44900	44899	44900	45318	45123	45235
	poly v-calls (of ∼23K)	1515	1514	1429	1404	1412	1404	1163	1163	1163	1233	1202	1229
	may-fail casts (of ∼2K)	1156	1155	1204	1089	1096	1094	727	616	621	879	744	766
	elapsed time (s)	81	478	117	406	**105**	126	532	1332	**359**	152	278	**135**
	sensitive var-points-to (M)	12.3	61.5	9.4	42.3	**7.6**	10.8	44.6	89.8	**32.3**	13.6	24.4	**11.5**
hsqldb	avg. objs per var	18.56	18.53	15.41	15.30	15.58	15.32	-	-	-	7.92	6.71	7.74
	edges (over ∼10K meths)	54619	54619	53726	53724	53730	53724	-	-	-	49421	49319	49421
	poly v-calls (of ∼26K)	1552	1552	1480	1479	1482	1479	-	-	-	1276	1263	1276
	may-fail casts (of ∼2K)	1360	1360	1385	1302	1320	1308	-	-	-	1031	923	948
	elapsed time (s)	90	332	218	1351	**183**	329	-	-	-	**195**	583	238
	sensitive var-points-to (M)	9.6	39.8	13.9	74.3	**9.6**	29.5	-	-	-	**13.7**	42.9	20.5
jython	avg. objs per var	20.64	20.57	18.21	18.01	18.19	18.09	-	-	-	8.55	7.18	8.30
	edges (over ∼8.5K meths)	50494	50480	49622	49614	49622	49614	-	-	-	43269	43138	43269
	poly v-calls (of ∼21K)	1525	1524	1448	1448	1453	1448	-	-	-	1268	1236	1268
	may-fail casts (of ∼1.9K)	1140	1140	1157	1087	1094	1092	-	-	-	909	822	840
	elapsed time (s)	88	401	119	375	**102**	138	-	-	-	731	1363	**676**
	sensitive var-points-to (M)	10.4	50.6	8.7	43.2	**6.7**	10.8	-	-	-	**52**	118.4	56.5
luindex	avg. objs per var	17.65	17.58	14.94	14.81	14.97	14.83	4.77	4.55	4.55	6.20	5.15	5.92
	edges (over ∼7.9K meths)	41992	41992	41103	41103	41111	41103	36580	36580	36580	36889	36796	36889
	poly v-calls (of ∼18K)	1180	1180	1119	1119	1122	1119	894	894	894	949	932	949
	may-fail casts (of ∼1.4K)	838	838	864	779	786	784	494	406	411	622	507	535
	elapsed time (s)	59	172	76	227	**70**	81	131	377	**105**	75	132	**73**
	sensitive var-points-to (M)	7.8	26.1	5.4	26.3	**4.1**	6.4	11.1	22.4	**7.2**	4.5	7.6	**3.5**
lusearch	avg. objs per var	18.64	18.47	15.71	15.57	15.79	15.60	4.71	4.49	4.50	6.13	5.10	5.86
	edges (over ∼8.4K meths)	45270	45270	44371	44365	44379	44371	39452	39446	39452	39763	39662	39763
	poly v-calls (of ∼19K)	1360	1360	1299	1299	1302	1299	1065	1065	1065	1122	1103	1122
	may-fail casts (of ∼1.5K)	939	939	961	874	884	880	506	410	416	662	537	568
	elapsed time (s)	63	187	89	279	**84**	95	183	464	**158**	76	137	**74**
	sensitive var-points-to (M)	8.7	28.5	6.2	30.3	**5.3**	7.2	13.2	26.3	**10**	4.2	7.8	**3.6**
pmd	avg. objs per var	19.94	19.82	17.36	17.22	17.37	17.24	4.87	4.68	4.68	6.35	5.28	6.10
	edges (over ∼9.2K meths)	49097	49097	48250	48250	48258	48250	43068	43067	43067	43401	43315	43400
	poly v-calls (of ∼21K)	1249	1249	1187	1187	1190	1187	937	937	937	988	976	988
	may-fail casts (of ∼2K)	1274	1274	1304	1215	1222	1220	844	752	757	1000	876	909
	elapsed time (s)	90	245	135	420	**128**	142	167	465	**145**	114	201	**113**
	sensitive var-points-to (M)	11.4	35.9	7.9	42.6	**6.9**	9.2	13.2	30.5	**10**	4.5	9.7	**3.9**
xalan	avg. objs per var	25.50	25.38	21.86	21.59	21.84	21.69	5.48	5.22	5.23	7.52	6.19	7.16
	edges (over ∼10.5K meths)	57168	57168	56412	56158	56404	56395	50148	50054	50054	50539	50432	50526
	poly v-calls (of ∼26K)	1976	1976	1920	1905	1921	1918	1619	1615	1615	1677	1660	1677
	may-fail casts (of ∼2K)	1213	1213	1236	1132	1140	1138	718	613	619	946	806	844
	elapsed time (s)	108	470	189	591	**161**	205	4521	3364	**1105**	168	299	**161**
	sensitive var-points-to (M)	14.5	59.8	15.5	67.5	**10.9**	18.2	166.6	171.7	**63.3**	10.2	17.4	**9**

Table 1: Precision and performance metrics for all benchmarks and analyses, grouped by relevance. In all cases *lower is better*. Dash (-) entries are for analyses that did not terminate in 90mins. The 4 precision metrics shown are the average size of points-to sets (how many heap objects are computed to be pointed-to per-var), the number of edges in the computed call-graph, the number of virtual calls whose target cannot be disambiguated by the analysis, and the number of casts that cannot be statically shown safe by the analysis. Reference numbers (e.g., total reachable casts in the program) are shown in parentheses in the metric's heading. These numbers change little per-analysis. Performance is shown as running time and size of context-sensitive var-points-to data (the main platform-independent internal complexity metric). Best performance numbers per-analysis-group are **in bold**.

an average points-to size of under 5. The difference in precision is also vividly shown in the "may-fail casts" metric: the 1call analysis cannot prove 2500 casts safe, while the U-2obj+H fails to prove safe just 915 casts (both numbers from a total of about 3.5K reachable casts—the exact number varies slightly due to method reachability variation per analysis).

- ***Uniform hybrid analyses.*** Recall that uniform hybrid analyses (U-1obj, U-2obj+H, U-2type+H) were defined to always keep a combination of object-sensitive and call-site-sensitive context. As a result, the analyses are more precise than their respective base analyses (1obj, 2obj+H, 2type+H), especially in the "may-fail casts" metric. However, this precision comes at great cost: uniform hybrid analyses are often 3x or more slower than their base analyses with twice as large, or more, context-sensitive points-to sets. U-1obj and U-2obj+H are plainly bad tradeoffs in the design space: for a slight increase in precision, the performance cost is heavy. U-2type+H is a bit more reasonable: it achieves more significant precision gains and its performance toll is often under 2x while still terminating comfortably for all our benchmarks. In fact, a surprising finding was that U-2type+H is a tempting alternative to 2obj+H for applications that need very high precision, given its good scalability.
- ***1obj hybrids.*** We presented two selective hybrids of a 1-object-sensitive analysis: S_A-1obj (which keeps either an allocation site or a call-site as context, but not both) and S_B-1obj (which always keeps an allocation site as context and occasionally adds a call-site to it). They both turn out to be interesting analyses from a practical standpoint. The former is consistently faster than the base 1obj analysis, with roughly similar precision and occasionally (for the "may-fail casts" metric) higher precision. The size of context-sensitive points-to sets also confirms that this is a "lighter" analysis that is likely to cost less in any context. The S_B-1obj analysis is always more precise than 1obj (as is statically guaranteed) for a slight extra cost. Indeed, S_B-1obj is a good approximation of the uniform hybrid analysis, U-1obj, in terms of precision, for a fraction (typically less than a third) of the cost.
- ***2obj+H hybrids.*** The selective hybrid idea yields even more dividends when applied to the very precise 2obj+H analysis. S-2obj+H is more precise than 2obj+H and only very slightly less precise than the uniform hybrid, U-2obj+H. In terms of performance, however, the analysis is typically well over 3 times faster than U-2obj+H, and significantly faster (an average of 53% speedup) than 2obj+H. This is interesting, given the practical value of 2obj+H, since it establishes a new sweet spot in the space of relatively scalable but highly precise analyses: S-2obj+H is both more precise than 2obj+H (especially for "may-fail casts") and substantially faster.
- ***2type+H hybrids.*** The 2type+H analysis variations are also highly interesting in practice. This is an analysis space that yields excellent precision relative to its low cost. There are few cases in which one might prefer some other inexpensive analysis over 2type+H given the combination of precision and competitive performance of the latter. As we saw, the uniform hybrid, U-2type+H, is an interesting tradeoff in this space. The selective hybrid, S-2type+H, also performs quite well. It is just as fast or slightly faster than the base analysis 2type+H, while also being more precise.

5. Related Work

We have discussed directly related work throughout the paper. Here we selectively mention a few techniques that, although not directly related to ours, offer alternative approaches to sweet spots in the precision/performance tradeoff.

Special-purpose combinations of context-sensitivity have been used in the past, but have required manual identification of classes to be treated separately (e.g., Java collection classes, or library factory methods). An excellent representative is the TAJ work for taint analysis of Java web applications [27]. In contrast, we have sought to map the space and identify interesting hybrids for general application of context-sensitivity, over the entire program.

The analyses we examined are context-sensitive but flow-insensitive. We can achieve several of the benefits of flow-sensitivity by applying the analysis on the static single assignment (SSA) intermediate form of the program. This is easy to do with a mere flag setting on the DOOP framework. However, the impact of the SSA transformation on the input is minimal. The default intermediate language used as input in DOOP (the Jimple representation of the Soot framework [28, 29]) is already close to SSA form, although it does not guarantee that every variable is strictly single-assignment without requesting it explicitly. Recent work by Lhoták and Chung [13] has shown that much of the benefit of flow-sensitivity derives from the ability to do strong updates of the points-to information. Lhoták and Chung then exploited this insight to derive analyses with similar benefit to a full flow-sensitive analysis at lower cost.

A demand-driven evaluation strategy reduces the cost of an analysis by computing only those results that are necessary for a client program analysis [10, 25, 26, 32]. This is a useful approach for client analyses that focus on specific locations in a program, but if the client needs results from the entire program, then demand-driven analysis is typically slower than an exhaustive analysis.

Reps [21] showed how to use the standard magic-sets optimization to automatically derive a demand-driven analysis from an exhaustive analysis (like ours). This optimization combines the benefits of top-down and bottom-up evaluation of logic programs by adding side-conditions to rules that limit the computation to just the required data.

An interesting recent approach to demand-driven analyses was introduced by Liang and Naik [15]. Their "pruning" approach 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 precise analysis is run, in order to establish the desired property.

6. Conclusions and Future Work

We presented a comprehensive map for the exploration of context combinations in points-to analysis, and used it to discover several interesting design points. Object-sensitivity and call-site-sensitivity had never been fruitfully combined in the past, although the idea is clearly tempting. We speculate that the reasons for the paucity of hybrid context-sensitivity results have been a) the difficulty of having a good enough model for the space of combinations and a convenient implementation to explore it; b) a belief that nothing fruitful will come out of such a combination, because call-site sensitivity incurs a high performance cost, which is more profitably spent on an extra level of object-sensitivity. The latter insight is mostly true, but only if one considers uniform hybrid analyses. As we saw, much of the benefit of call-site and object-sensitive hybrids comes from allowing the context to vary between pure object-sensitive and extended. The result of our work has been new sweet spots, in both precision and performance, for some of the most practically relevant analysis variations.

There are several interesting directions for further work that open up. First, our model gives the ability for further experimentation, e.g., with deeper-context analyses. Furthermore, it is interesting to examine if a hybrid context should perhaps change form more aggressively. The **Merge** and **MergeStatic** functions could examine the context passed to them as argument and create different kinds of contexts in return. For instance, the context of a statically called method could have a different form (e.g., more elements) for a call made inside another statically called method vs. a call made in a virtual method. Similarly, objects could have different context, via the **Record** function, depending on the context form of their allocating method. To explore this space without blind guessing, one needs to understand what programming patterns are best handled by hybrid contexts and how. For deep contexts this remains a challenge, as it is hard to reason about how context elements affect precision. (E.g., past work had to offer involved arguments for why the allocator object of the receiver object of a method is a better context element than the caller object [24].) This challenge is, however, worth addressing for the next level of benefit in context-sensitive points-to analysis.

Acknowledgments

We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant and a European Research Council Starting/Consolidator grant; and by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award. We thank the anonymous reviewers, who offered several valuable suggestions, and LogicBlox Inc. for providing our Datalog engine, as well as technical and material support.

References

[1] K. Ali and O. Lhoták. Application-only call graph construction. In *European Conf. on Object-Oriented Programming (ECOOP)*, 2012.

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

[3] M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Conf. on Programming Language Design and Implementation (PLDI)*, 2003.

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

[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)*, 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 *European Conf. on Object-Oriented Programming (ECOOP)*, 2006.

[8] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Conf. on Programming Language Design and Implementation (PLDI)*, 2007.

[9] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Symposium on Principles of Programming Languages (POPL)*, 2009.

[10] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Conf. on Programming Language Design and Implementation (PLDI)*, 2001.

[11] 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)*, 2005.

[12] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, 2006.

[13] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Symposium on Principles of Programming Languages (POPL)*, 2011.

[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] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *Conf. on Programming Language Design and Implementation (PLDI)*, 2011.

[16] 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, 2012.

[17] 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)*, 2010.

[18] 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)*, 2002.

[19] 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.

[20] T. Reps. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, 1994.

[21] T. W. Reps. Solving demand versions of interprocedural analysis problems. In *Int. Conf. on Compiler Construction (CC)*, 1994.

[22] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis*, 1981.

[23] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 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)*, 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)*, 2006.

[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)*, 2005.

[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)*, 2009.

[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)*, 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*, 1999.

[30] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In *APLAS*, 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)*, 2004.

[32] X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *Symposium on Principles of Programming Languages (POPL)*, 2008.