Context-sensitive points-to analysis: is it worth it?*

Ondřej Lhoták[1,2] and Laurie Hendren[2]
olhotak@uwaterloo.ca hendren@sable.mcgill.ca

[1] School of Computer Science, University of Waterloo, Waterloo, ON, Canada
[2] School of Computer Science, McGill University, Montreal, QC, Canada

Abstract. We present the results of an empirical study evaluating the precision of subset-based points-to analysis with several variations of context sensitivity on Java benchmarks of significant size. We compare the use of call site strings as the context abstraction, object sensitivity, and the BDD-based context-sensitive algorithm proposed by Zhu and Calman, and by Whaley and Lam. Our study includes analyses that context-sensitively specialize only pointer variables, as well as ones that also specialize the heap abstraction. We measure both characteristics of the points-to sets themselves, as well as effects on the precision of client analyses. To guide development of efficient analysis implementations, we measure the number of contexts, the number of distinct contexts, and the number of distinct points-to sets that arise with each context sensitivity variation. To evaluate precision, we measure the size of the call graph in terms of methods and edges, the number of devirtualizable call sites, and the number of casts statically provable to be safe.

The results of our study indicate that object-sensitive analysis implementations are likely to scale better and more predictably than the other approaches; that object-sensitive analyses are more precise than comparable variations of the other approaches; that specializing the heap abstraction improves precision more than extending the length of context strings; and that the profusion of cycles in Java call graphs severely reduces precision of analyses that forsake context sensitivity in cyclic regions.

1 Introduction

Does context sensitivity significantly improve precision of interprocedural analysis of object-oriented programs? It is often suggested that it could, but lack of scalable implementations has hindered thorough empirical verification of this intuition.

Of the many context sensitive points-to analyses that have been proposed (e.g. [1, 4, 8, 11, 17–19, 25, 28–31]), which improve precision the most? Which are most effective for specific client analyses, and for specific code patterns? For which variations are we likely to find scalable implementations? Before devoting resources to finding efficient implementations of specific analyses, we should have empirical answers to these questions.

This study aims to provide these answers. Recent advances in the use of Binary Decision Diagrams (BDDs) in program analysis [3, 12, 29, 31] have made context sensitive analysis efficient enough to perform an empirical study on benchmarks of significant size. Using the JEDD system [14], we have implemented three different families of context-sensitive points-to analysis, and we have measured their precision in terms of several client analyses. Specifically, we compared the use of call-site strings as the context abstraction, object sensitivity [17, 18], and the algorithm proposed by Zhu and Calman [31]

* This work was supported, in part, by NSERC and an IBM Ph.D. Fellowship.

and Whaley and Lam [29] (hereafter abbreviated ZCWL). Within each family, we evaluated the effect of different lengths of context strings, and of context-sensitively specializing the heap abstraction. In our study, we compared the relative precision of analyses both quantitatively, by computing summary statistics about the analysis results, and qualitatively, by examining specific code patterns for which a given analysis variation produces better results than other variations.

Context-sensitive analyses have been associated with very large numbers of contexts. We wanted to also determine how many contexts each variation of context sensitivity actually generates, how the number of contexts relates to the precision of the analysis results, and how likely it is that scalable context-sensitive representations are feasible. These measurements can be done directly on the BDD representation.

Our results show that although the effect on precision depends on the client analysis, the benefits of context sensitivity are very significant for some analyses, particularly cast safety analysis. We also show that object-sensitivity consistently improves precision most compared to the other variations studied, and that modelling heap objects with context does significantly improve precision.

The remainder of this paper is organized as follows. In Section 2, we provide background about the variations of context sensitivity that we have studied. In Section 3, we list the benchmarks included in our study. We discuss the number of contexts and its implications on precision and scalability in Section 4. In Section 5, we examine the effects of context sensitivity on the precision of the call graph. We evaluate opportunities for static resolution of virtual calls in Section 6. In Section 7, we measure the effect of context sensitivity on cast safety analysis. We briefly survey related work in Section 8. Finally, we draw conclusions from our experimental results in Section 9.

2 Background

Like any static analysis, a points-to analysis models the possible run-time features of the program using some chosen static abstraction. A context-sensitive points-to analysis requires an abstraction of pointer targets, pointers, and method invocations. We will denote these three abstractions $\mathcal{O}$, $\mathcal{P}$, and $\mathcal{I}$, respectively. Whenever it is possible for a run-time pointer p to point to the run-time target o, the may-point-to relation computed by the analysis must contain the fact $\mathcal{O}(o) \in pt(\mathcal{P}(p))$. The specific choice of static abstraction is a key determining factor of the precision of the analysis, and this paper compares several different abstractions.

Pointer Target Abstraction: In Java, the target of a pointer is always a dynamically allocated object. A popular abstraction for a pointer target is the program statement at which the object was allocated. We will write this abstraction as $\mathcal{O}^{as}$.

Pointer Abstraction: Each run-time pointer corresponds to either some local variable or some object field in the program. Pointers corresponding to local variables are often statically abstracted by the local variable; we will write this abstraction as $\mathcal{P}^{var}$. For pointers corresponding to fields, we will consider only the field-sensitive abstraction in this paper, because it is more precise than other alternatives (described, for example, in [13, 23]). The field-sensitive abstraction $\mathcal{P}^{fs}(o.f)$ of the field f of run-time object o is the pair $[\mathcal{O}(o), f]$, where $\mathcal{O}(o)$ is our chosen static abstraction of the run-time object o.

Method Invocation (Context) Abstraction: Because different invocations of a method may have different behaviours, it may be useful to distinguish some of them. A context is

a static abstraction of a method invocation; an analysis distinguishes invocations if their abstract contexts are different. In this paper, we compare two families of invocation abstraction (also called context abstraction), call sites [24, 25] and receiver objects [17, 18]. In call-site context sensitivity, the context $\mathcal{I}^{cs}(i)$ of an invocation i is the program statement (call site) from which the method was invoked. In receiver-object context sensitivity, the context of an invocation i is the static abstraction of the object on which the method is invoked. That is, $\mathcal{I}^{ro}(i) = \mathcal{O}(o)$, where o is the run-time object on which the method was invoked.

In either case, the context abstraction can be made even finer by using a string of contexts corresponding to the invocation frames on the run-time invocation stack [18, 24]. That is, having chosen a base abstraction $\mathcal{I}^{base}$, we can define $\mathcal{I}^{string}(i)$ to be $[\mathcal{I}^{base}(i), \mathcal{I}^{base}(i_2), \mathcal{I}^{base}(i_3), \ldots]$, where i_j is the j'th top-most invocation on the stack during the invocation i (so $i = i_1$). Since the maximum height of the stack is unbounded, the analysis must somehow ensure that the static abstraction is finite. A simple, popular technique is to limit the length of each context string to at most a fixed number k. A different technique is used by the ZCWL algorithm. It does not limit the length of a context string, but it excludes from the context string all contexts corresponding to call edges that are part of a cycle in the context-insensitive call graph. Thus, the number of contexts is bounded by the number of acyclic paths in the call graph, which is finite.

Orthogonal to the choice of context abstraction is the choice of which pointers and objects to model context-sensitively. That is, having chosen a basic context-insensitive pointer abstraction $\mathcal{P}^{ci}$ and a context abstraction $\mathcal{I}$, we can model a run-time pointer p context-sensitively by defining $\mathcal{P}(p)$ to be $[\mathcal{I}(i_p), \mathcal{P}^{ci}(p)]$, where i_p is the method invocation in which p occurs, or context-insensitively by defining $\mathcal{P}(p)$ to be $\mathcal{P}^{ci}(p)$. Similarly, if we have chosen the allocation site abstraction $\mathcal{O}^{as}$ as the basic abstraction for objects, we can model each object o context-sensitively by defining $\mathcal{O}(o)$ to be $[\mathcal{I}(i_o), \mathcal{O}^{as}(o)]$, where i_o is the method invocation during which o was allocated, or context-insensitively by defining $\mathcal{O}(o)$ to be $\mathcal{O}^{as}(o)$.

In the tables in the rest of this paper, we report results for the following variations of points-to analyses. In tables reporting call graph information, the "CHA" column reports baseline numbers obtained using Class Hierarchy Analysis [6]. The "insens." column of each table is a context-insensitive points-to analysis that does not distinguish different invocations of any method. The "object-sensitive" columns are analyses using receiver objects as the context abstraction, while the "call site" columns are analyses using call sites as the context abstraction. Within each of these two sections, in the 1, 2, and 3 columns, pointers are modelled with context strings of maximum length 1, 2, and 3, but pointer targets are modelled context-insensitively. In the 1H columns, both pointers and pointer targets are modelled with context strings of receiver objects or call sites of maximum length 1. The "ZCWL" column is the ZCWL algorithm, which uses call sites as the context abstraction, and allows context strings of arbitrary length. The ZCWL algorithm models pointers with context but pointer targets without context.

In an analysis of an object-oriented language such as Java, there is a cyclic dependency between call graph construction and points-to analysis. In all variations except the ZCWL algorithm, we constructed the call graph on-the-fly during the points-to analysis, since this maintains maximum precision. The ZCWL algorithm requires a context-insensitive call graph to be constructed before it starts, which it then makes context-

sensitive, and uses to perform the points-to analysis. For this purpose, we used the call graph constructed by the context-insensitive analysis in the "insens." column.

Interested readers can find additional information about the analysis variations, as well as a detailed presentation of the analysis implementation, in [12, Chapter 4].

3 Benchmarks

	Total number of		Executed methods	
Benchmark	classes	methods	app.	+lib.
compress	41	476	56	463
db	32	440	51	483
jack	86	812	291	739
javac	209	2499	778	1283
jess	180	1482	395	846
mpegaudio	88	872	222	637
mtrt	55	574	182	616
soot-c	731	3962	1055	1549
sablecc-j	342	2309	1034	1856
polyglot	502	5785	2037	3093
antlr	203	3154	1099	1783
bloat	434	6125	138	1010
chart	1077	14966	854	2790
jython	270	4915	1004	1858
pmd	1546	14086	1817	2581
ps	202	1147	285	945

Table 1. Benchmarks

We performed our study on programs from the SpecJVM 98 benchmark suite [26], the DaCapo benchmark suite, version beta050224 [5], and the Ashes benchmark suite [27], as well as on the Polyglot extensible Java front-end [20], as listed in Table 1. Most of these benchmarks have been used in earlier evaluations of interprocedural analyses for Java. The middle section of the table shows the total number of classes and methods comprising each benchmark. These numbers exclude the Java standard library (which is required to run the benchmark), but include all other libraries that must accompany the benchmark for it to run successfully. All of the measurements in this paper were done with version 1.3.1_01 of the Sun standard library.[3] The right-most section of the table shows the number of distinct methods that are executed in a run of the benchmark (measured using the *J tool [7]), both excluding and including methods of the Java standard library, in the columns labelled "app." and "+lib.", respectively. About 400 methods of the standard library are executed even for the smallest benchmarks for purposes such as class loading; some of the larger benchmarks make heavier use of the library.

4 Number of Contexts

Context-sensitive analysis is often considered intractable mainly because, if contexts are propagated from every call site to every called method, the number of resulting context strings grows exponentially in the length of the call chains. The purpose of this section is to shed some light on two issues. First, of the large numbers of contexts, how many

[3] Studying other standard library versions requires models of their native methods. We aim to write such models for a more recent version as future work.

are actually useful in improving analysis results? Second, why can BDDs represent such seemingly large numbers of contexts, and how much hope is there that they can be represented with more traditional techniques?

4.1 Total number of contexts

We begin by comparing the number of contexts that appear in the context-sensitive points-to relation when the analysis is performed with the different context abstractions. For this measurement, we treat the method invoked as part of the context. For example, suppose we are using abstract receiver objects as the context abstraction; if two different methods are called on the same receiver, we count them as two separate contexts, since they correspond to two necessarily distinct invocations. In other words, we are counting method-context pairs, rather than just contexts.

The measurements of the total numbers of contexts are shown in Table 2. Each column lists the number of contexts produced by one of the variations of context-sensitive analysis described in Section 2. The column labelled "insens." shows the absolute number of contexts (which is also the number of methods, since in a context-insensitive analysis, every method has exactly one context). All the other columns, rather than showing the absolute number of contexts, which would be very large, instead show the number of contexts as a multiple of the "insens." column (*i.e.* they show the average number of contexts per method). For example, for the compress benchmark, the total number of 1-object-sensitive contexts is $2596 \times 13.7 = 3.56 \times 10^4$. The empty spots in the table (and other tables throughout this paper) indicate configurations in which the analysis did not complete in the available memory, despite being implemented using BDDs. We allowed the BDD library to allocate a maximum of 41 million BDD nodes (820 million bytes).

		object-sensitive				call site				
Benchmark	insens.	1	2	3	1H	1	2	1H	ZCWL	(max. k)
compress	2596	13.7	113	1517	13.4	6.5	237	6.5	2.9×10^4	(21)
db	2613	13.7	115	1555	13.4	6.5	236	6.5	7.9×10^4	(22)
jack	2869	13.8	156	1872	13.2	6.8	220	6.8	2.7×10^7	(45)
javac	3780	15.8	297	13289	15.6	8.4	244	8.4		(41)
jess	3216	19.0	305	5394	18.6	6.7	207	6.7	6.1×10^6	(24)
mpegaudio	2793	13.0	107	1419	12.7	6.3	221	6.3	4.4×10^5	(31)
mtrt	2738	13.3	108	1447	13.1	6.6	226	6.6	1.2×10^5	(26)
soot-c	4837	11.1	168	4010	10.9	8.2	198	8.2		(39)
sablecc-j	5608	10.8	116	1792	10.5	5.5	126	5.5		(55)
polyglot	5616	11.7	149	2011	11.2	7.1	144	7.1	10130	(22)
antlr	3897	15.0	309	8110	14.7	9.6	191	9.6	4.8×10^9	(39)
bloat	5237	14.3	291		14.0	8.9	159	8.9	3.0×10^8	(26)
chart	7069	22.3	500		21.9	7.0	335			(69)
jython	4401	18.8	384		18.3	6.7	162	6.7	2.1×10^{15}	(72)
pmd	7219	13.4	283	5607	12.9	6.6	239	6.6		(55)
ps	3874	13.3	271	24967	13.1	9.0	224	9.0	2.0×10^8	(29)

Note: columns after the second column show multiples of the context-insensitive number.

Table 2. Total number of abstract contexts

The large numbers of contexts explain why an analysis that represents each context explicitly cannot scale to the programs that we analyze here. While a 1-call-site-sensitive analysis must store and process 6 to 9 times more data than a context-insensitive analysis, the ratio grows to 1500 or more times for a 3-object-sensitive analysis.

The ZCWL algorithm essentially performs a k-CFA analysis in which k is the maximum call depth in the original call graph after merging strongly connected components (shown in parentheses in the ZCWL column). Because k is different for each benchmark, the number of contexts is much more variable than in the other variations of context sensitivity. On the javac, soot-c, sablecc-j, chart, and pmd benchmarks, the algorithm failed to complete in the available memory.

4.2 Equivalent contexts

Next, we consider that many of the large numbers of abstract contexts are equivalent in the sense that the points-to relations computed in many of the abstract contexts are the same. More precisely, we define two method-context pairs, (m_1, c_1) and (m_2, c_2) to be ***equivalent*** if $m_1 = m_2$, and for every local pointer variable p in the method, the points-to set of p is the same in both contexts c_1 and c_2.

When two contexts are equivalent, there is no point in distinguishing them, because the resulting points-to relation is independent of the context. In this sense, the number of equivalence classes of method-context pairs reflects how worthwhile context sensitivity is in improving the precision of points-to sets.

The measurements of the number of equivalence classes of contexts are shown in Table 3. Again, the "insens." column shows the actual number of equivalence classes of contexts, while the other columns give a multiple of the "insens." number (*i.e.* the average number of equivalence classes per method).

		object-sensitive				call site			
Benchmark	insens.	1	2	3	1H	1	2	1H	ZCWL
compress	2597	8.4	9.9	11.3	12.1	2.4	3.9	4.9	3.3
db	2614	8.5	9.9	11.4	12.1	2.4	3.9	5.0	3.3
jack	2870	8.6	10.2	11.6	11.9	2.4	3.9	5.0	3.4
javac	3781	10.4	17.7	33.8	14.3	2.7	5.3	5.4	
jess	3217	8.9	10.6	12.0	13.9	2.6	4.2	5.0	3.9
mpegaudio	2794	8.1	9.4	10.8	11.5	2.4	3.8	4.8	3.3
mtrt	2739	8.3	9.7	11.1	11.8	2.5	4.0	4.9	3.4
soot-c	4838	7.1	13.7	18.4	9.8	2.6	4.2	4.8	
sablecc-j	5609	6.9	8.4	9.6	9.5	2.3	3.6	3.9	
polyglot	5617	7.9	9.4	10.8	10.2	2.4	3.7	4.7	3.3
antlr	3898	9.4	12.1	13.8	13.2	2.5	4.1	5.2	4.3
bloat	5238	10.2	44.6		12.9	2.8	4.9	5.2	6.7
chart	7070	10.0	17.4		18.2	2.7	4.8		
jython	4402	9.9	55.9		15.6	2.5	4.3	4.6	4.0
pmd	7220	7.6	14.6	17.0	11.0	2.4	4.2	4.2	
ps	3875	8.7	9.9	11.0	12.0	2.6	4.0	5.2	4.4

Note: columns after the second column show multiples of the context-insensitive number.

Table 3. Number of equivalence classes of abstract contexts

The relatively small size of these numbers compared to the total numbers of contexts in Table 2 explains why a BDD can effectively represent the analysis information, since it automatically merges the representation of equal points-to relations, so each distinct relation is only represented once. If we had some idea before designing an analysis which abstract contexts are likely to be equivalent, we could define a new context abstraction in which these equivalent contexts are merged. Each equivalence class of old abstract

contexts would be represented by a single new abstract context. With such a context abstraction, the context-sensitive analysis could be implemented without requiring BDDs.

It is interesting that in the 1-, 2-, and 1H-object-sensitive analysis, the number of equivalence classes of contexts is generally about 3 times as high as in the corresponding 1-, 2-, and 1H-call-site-string analysis. This indicates that receiver objects better partition the space of concrete calling contexts that give rise to distinct points-to relations. That is, if at run time, the run-time points-to relation is different in two concrete calls to a method, it is more likely that the two calls will correspond to distinct abstract contexts if receiver objects rather than call sites are used as the context abstraction. This observation leads us to hypothesize that object-sensitive analysis should be more precise than call-site-string analysis; we will see more direct measurements of precision in upcoming sections.

In both object-sensitive and call-site-string analyses, making the context string longer increases the number of equivalence classes of contexts by only a small amount, while it increases the absolute number of contexts much more significantly. Therefore, increasing the length of the context string is unlikely to result in a large improvement in precision, but will significantly increase analysis cost.

It was initially rather surprising that in the analysis using the ZCWL algorithm, the number of equivalence classes of abstract contexts is so small, often even smaller than in the 2-call-site-sensitive analysis. The algorithm essentially performs a k-CFA analysis, where k is the maximum call depth in the original call graph; k is always much higher than 2. The number of equivalence classes of contexts when using the ZCWL algorithm is small because the algorithm merges strongly connected components (SCCs) in the call graph, and models all call edges in each such component in a context-insensitive way. In contrast, the 2-call-site-sensitive analysis models all call edges context-sensitively, including those in SCCs. Indeed, a very large number of methods are part of some SCC. The initial call graph for each of our benchmarks contains a large SCC of 1386 to 2926 methods, representing 36% to 53% of all methods in the call graph. In particular, this SCC always includes many methods for which context-sensitive analysis would be particularly useful, such as the methods of the String class and the standard collections classes. These methods are used extensively within the Java standard library, and contain many calls to each other. We examined this large SCC and found many distinct cycles; there was no single method that, if removed, would break the component. In summary, the reason for the surprisingly small number of equivalence classes of abstract contexts when using the ZCWL algorithm is that it models a large part of the call graph context-insensitively.

4.3 Distinct points-to sets

Finally, we measure the number of distinct points-to sets that appear in the points-to analysis result. This number is an indication of how difficult it would be to efficiently represent the context-sensitive points-to sets in a non-BDD-based analysis implementation, assuming there was already a way to represent the contexts themselves. An increase in the number of distinct points-to sets also suggests an increase in precision, but the connection is very indirect [10, Section 3.2]. We therefore present the number of distinct points-to sets primarily as a measure of analysis cost, and provide more direct measurements of the precision of clients of the analysis later in this paper. In traditional, context-insensitive, subset-based points-to analyses, the representation of the points-to sets often makes up most of the memory requirements of the analysis. If the traditional analysis stores points-to sets using shared bit-vectors as suggested by Heintze [9], each distinct

points-to set need only be stored once. Therefore, the number of distinct points-to sets approximates the space requirements of such a traditional representation.

The measurements of the number of distinct points-to sets arising with each context abstraction are shown in Table 4. In this table, all numbers are the absolute count of distinct points-to sets, not multiples of the "insens." column.

		object-sensitive				call site			
Benchmark	insens.	1	2	3	1H	1	2	1H	ZCWL
compress	3178	3150	3240	3261	34355	3227	3125	38242	3139
db	3197	3170	3261	3283	34637	3239	3133	38375	3173
jack	3441	3411	3507	3527	37432	3497	3377	40955	3541
javac	4346	4367	4579	4712	55196	4424	4303	54866	
jess	3834	4433	4498	4514	51452	4589	4426	42614	4644
mpegaudio	4228	4179	4272	4293	36563	4264	4157	67565	4175
mtrt	3349	3287	3377	3396	35154	3387	3263	38758	3282
soot-c	4683	4565	4670	4657	45974	4722	4550	52937	
sablecc-j	5753	5777	5895	5907	52993	5875	5694	59748	
polyglot	5591	5556	5829	5925	50587	5682	5516	59837	5575
antlr	4520	5259	5388	5448	54942	4624	4535	54176	4901
bloat	5337	5480	5815		55309	5452	5342	49230	6658
chart	9608	9914	10168		233723	9755	9520		
jython	4669	5111	5720		74297	4968	4857	46280	8587
pmd	7368	7679	7832	7930	94403	7671	7502	103990	
ps	4610	4504	4639	4672	47244	4656	4521	58513	4802

Table 4. Total number of distinct points-to sets in points-to analysis results

The numbers of distinct points-to sets are fairly constant in most of the analysis variations, including object-sensitive analyses, call-site-string analyses, and the analysis using the ZCWL algorithm. Therefore, in a traditional points-to analysis implemented using shared bit-vectors, representing the individual points-to sets should not be a source of major difficulty even in a context-sensitive analysis. Future research in traditional implementations of context-sensitive analyses should therefore be directed more at the problem of efficiently representing the contexts, rather than representing the points-to sets.

However, when abstract heap objects are modelled context-sensitively, the elements of each points-to set are pairs of abstract object and context, rather than simply abstract objects, and the number of distinct points-to sets increases about 11-fold. In addition, it is likely that the points-to sets themselves are significantly larger. Therefore, in order to implement such an analysis without using BDDs, it would be worthwhile to look for an efficient way to represent points-to sets of abstract objects with context.

5 Call Graph

We now turn our attention to the effect of context sensitivity on call graph construction. For the purposes of comparison, we have constructed context-sensitive call graphs, projected away their contexts, and measured differences in their context-insensitive projections. We adopted this methodology because context-sensitive call graphs using different context abstractions are not directly comparable. Each node in the graph represents a pair of method and abstract context, but the set of possible abstract contexts is different in each context variation. In the context-insensitive projection, each node is simply a method, so the projections are directly comparable. The context-insensitive projection

preserves the set of methods reachable from the program entry points, as well as the set of possible targets of each call site in the program; it is these sets that we measure. The set of reachable methods is particularly important because any conservative interprocedural analysis must analyze all of these methods, so a small set of reachable methods reduces the cost of other interprocedural analyses.

We have not included the ZCWL algorithm in our study of call graph construction, because the context-insensitive projection of the context-sensitive call graph that it produces is the same as the context-insensitive call graph that we originally give it as input.

5.1 Reachable methods

Table 5 shows the number of methods reachable from the program entry points when constructing the call graph using different variations of context sensitivity, excluding methods from the standard Java library. In Table 5 and all subsequent tables in this paper, the most precise entry for each benchmark has been highlighted in bold. In the case of a tie, the most precise entry that is least expensive to compute has been highlighted.

			object-sensitive				call site			actually
Benchmark	CHA	insens.	1	2	3	1H	1	2	1H	executed
compress	90	**59**	59	59	59	59	59	59	59	56
db	95	65	**64**	64	64	64	65	64	65	51
jack	348	317	**313**	313	313	313	316	313	316	291
javac	1185	1154	**1147**	1147	1147	1147	1147	1147	1147	778
jess	683	630	629	629	629	**623**	629	629	629	395
mpegaudio	306	255	**251**	251	251	251	251	251	251	222
mtrt	217	189	**186**	186	186	186	187	187	187	182
soot-c	2395	2273	**2264**	2264	2264	2264	2266	2264	2266	1055
sablecc-j	1904	1744	1744	1744	1744	**1731**	1744	1744	1744	1034
polyglot	2540	2421	2419	2419	2419	**2416**	2419	2419	2419	2037
antlr	1374	**1323**	1323	1323	1323	1323	1323	1323	1323	1099
bloat	2879	2464	**2451**	2451		2451	2451	2451	2451	138
chart	3227	2081	2080	2080		**2031**	2080	2080		854
jython	2007	1695	1693	1693		**1683**	1694	1693	1694	1004
pmd	4997	4528	4521	4521	4521	**4509**	4521	4521	4521	1817
ps	840	835	835	835	835	**834**	835	835	835	285

Table 5. Number of reachable benchmark (non-library) methods in call graph

For the simple benchmarks like compress and db, the context-insensitive call graph is already quite precise (compared to the dynamic behaviour), and any further improvements due to context sensitivity are relatively small. For the more significant benchmarks, call graph construction benefits slightly from 1-object sensitivity. The largest difference is 13 methods, in the bloat benchmark. All of these methods are visit methods in an implementation of the visitor design pattern, in the class AscendVisitor. This class traverses a parse tree from a starting node upwards toward the root of the tree, visiting each node along the way. Some kinds of nodes have no descendants that are ever the starting node of a traversal, so the visit methods of these nodes can never be called. However, in order to prove this, an analysis must analyze the visitor dispatch method context-sensitively in order to keep track of the kind of node from which it was called. Therefore, a context-insensitive analysis fails to show that these visit methods are unreachable.

In jess, sablecc-j, polyglot, chart, jython, pmd, and ps, modelling abstract heap objects object-sensitively further improves the precision of the call graph. In the sablecc-j benchmark, 13 additional methods are proved unreachable. The benchmark includes an implementation of maps similar to those in the standard library. The maps are instantiated in a number of places, and different kinds of objects are placed in the different maps. Methods such as toString() and equals() are called on some maps but not others. Calling one of the methods on a map causes it to be called on all elements of the map. Therefore, these methods are called on some kinds of map elements, but not others. However, the map elements are kept in generic map entry objects, which are allocated at a single point in the map code. When abstract heap objects are modelled without context, all map entries are modelled by a single abstract object, and the contents of all maps are conflated. When abstract heap objects are modelled with context, the map entries are treated as separate objects depending on which map they were created for. Note that distinguishing the map entries requires receiver objects to be used as context, rather than call-site strings. The code that allocates a new entry is in a method that is always called from the same call site, in another method of the map class. In general, although modelling abstract heap objects with context improved the call graph for some benchmarks in an object-sensitive analysis, it never made any difference in analyses using call-site strings as the context abstraction (*i.e.* the 1-call-site and 1H-call-site columns are the same).

Overall, object-sensitive analysis results in slightly smaller call graphs than call-site-string analysis. The 1-object-sensitive call graph is never larger than the 1-call-site-sensitive call graph, and it is smaller on db, jack, mtrt, soot-c, and jython. On the db, jack, and jython benchmarks, the call-site-sensitive call graph can be made as small as the 1-object-sensitive call graph, but it requires 2-call-site rather than 1-call-site analysis.

Even the most precise context-sensitive analyses produce a much bigger call graph than the dynamic one, shown in the rightmost column of the table. This difference is largely due to unused but complicated features of the Java Runtime Environment (such as network class loading and Jar File signing) which are controlled by external configuration parameters unknown to the analysis.

5.2 Call edges

Table 6 shows the size of the call graph in terms of call edges rather than reachable methods. Only call edges originating from a benchmark (non-library) method are counted.

In general, context sensitivity makes little difference to the size of the call graph when measured this way, with one major exception. In the sablecc-j benchmark, the number of call edges is 17925 in a context-insensitive analysis, but only 5175 in a 1-object-sensitive analysis. This could make a significant difference to the cost of a client analysis whose complexity depends on the number of edges in the call graph. The large difference is caused by the following pattern of code. The sablecc-j benchmark contains code to represent a parse tree, with many different kinds of nodes. Each kind of node implements a method called removeChild(). The code contains a large number of calls of the form this.getParent().removeChild(this). In a context-insensitive analysis, getParent() is found to possibly return any of hundreds of possible kinds of nodes. Therefore, each of these many calls to removeChild(this) results in hundreds of call graph edges. However, in a context-sensitive analysis, getParent() is analyzed in the context of the this pointer. For each kind of node, there is a relatively small number of kinds of nodes that can be its parent. Therefore, in a given context, getParent() is found to return only a small number

Benchmark	CHA	insens.	object-sensitive 1	2	3	1H	call site 1	2	1H	actually executed
compress	456	**270**	270	270	270	270	270	270	270	118
db	940	434	**427**	427	427	427	434	427	434	184
jack	1936	1283	1251	1251	1251	**1250**	1276	1251	1276	833
javac	13146	10360	**10296**	10296	10296	10296	10318	10301	10318	2928
jess	4700	3626	3618	3618	3618	**3571**	3618	3618	3618	919
mpegaudio	1182	858	**812**	812	812	812	812	812	812	400
mtrt	925	761	**739**	739	739	739	746	746	746	484
soot-c	20079	14611	14112	14112	14112	**13868**	14185	14112	14185	2860
sablecc-j	24283	17925	5175	5140	5140	**5072**	5182	5140	5182	2326
polyglot	19898	11768	11564	11564	11564	**11374**	11566	11566	11566	5440
antlr	10769	**9553**	9553	9553	9553	9553	9553	9553	9553	4196
bloat	36863	18586	18143	18143		**17722**	18166	18143	18166	477
chart	24978	9526	9443	9443		**9178**	9443	9443		2166
jython	13679	9382	9367	9367		**9307**	9367	9365	9367	2898
pmd	29401	18785	18582	18582	18580	**18263**	18601	18599	18601	3879
ps	13610	11338	11292	11292	11292	**10451**	11298	11292	11298	705

Table 6. Number of call edges in call graph originating from a benchmark (non-library) method

of kinds of parent node, so each call site of removeChild() adds only a small number of edges to the call graph.

6 Virtual Call Resolution

Table 7 shows the number of virtual call sites for which the call graph contains more than one potential target method. Call sites with at most one potential target method can be converted to cheaper static instead of virtual calls, and they can be inlined, possibly enabling many other optimizations. Therefore, an analysis that proves that call sites are not polymorphic can be used to significantly improve run-time performance.

Benchmark	CHA	insens.	object-sensitive 1	2	3	1H	call site 1	2	1H
compress	16	**3**	3	3	3	3	3	3	3
db	36	5	**4**	4	4	4	5	4	5
jack	474	25	23	23	23	**22**	24	23	24
javac	908	737	**720**	720	720	720	720	720	720
jess	121	**45**	45	45	45	45	45	45	45
mpegaudio	40	27	**24**	24	24	24	24	24	24
mtrt	20	9	**7**	7	7	7	8	8	8
soot-c	1748	983	**913**	913	913	913	938	913	938
sablecc-j	722	450	325	325	325	**301**	380	325	380
polyglot	1332	744	592	592	592	**585**	592	592	592
antlr	1086	**843**	843	843	843	843	843	843	843
bloat	2503	1079	962	962		**961**	962	962	962
chart	2782	254	235	235		**214**	235	235	
jython	646	347	347	347		**346**	347	347	347
pmd	2868	1224	1193	1193	1193	**1163**	1205	1205	1205
ps	321	304	303	303	303	**300**	303	303	303

Table 7. Total number of potentially polymorphic call sites in benchmark (non-library) code

In the benchmarks written in an object-oriented style, notably javac, soot-c, sablecc-j, polyglot, bloat, and pmd, many more call sites can be devirtualized using object-sensitive analysis than context-insensitive analysis. In some cases, call-site-string analysis gives the same improvement, but never any more, and in soot-c and sablecc-j, the improvement from 1-object-sensitive analysis is much greater than from 1-call-site string analysis.

In sablecc-j, there are three sets of call sites that can be devirtualized using context-sensitive analysis. Any context-sensitive analysis is sufficient to devirtualize the first set of call sites. Devirtualization of the second set of call sites requires an object-sensitive analysis; an analysis using call sites as the context abstraction cannot prove them to be monomorphic. Devirtualization of the third set of call sites not only requires an object-sensitive analysis, but it also requires that abstract heap objects be modelled with context.

The first set of call sites are the calls to the removeChild() method mentioned in Section 5.2. Object sensitivity reduces the number of potential target methods at each of these call sites. At many of them, it reduces the number down to one, so the calls can be devirtualized. The same improvement is obtained with call-site-string context sensitivity.

The second set of call sites are calls to methods of iterators over lists. The sablecc-j benchmark contains several implementations of lists similar to those in the standard Java library. A call to iterator() on any of these lists invokes iterator() on the AbstractList superclass, which in turn invokes the listIterator() method specific to each list. The actual kind of iterator that is returned depends on which listIterator() was invoked, which in turn depends on the receiver object of the call to iterator(); it is independent of the call site of listIterator(), which is always the same site in iterator(). Therefore, calls to hasNext() and next() on the returned iterator can be devirtualized only with an object-sensitive analysis.

The third set of call sites are calls to methods such as toString() and equals() on objects stored in maps. As we explained in Section 5.1, object-sensitive modelling of abstract heap objects is required to distinguish the internal map entry objects in each separate use of the map implementation. The map entry objects must be distinguished in order to distinguish the objects that are stored in the maps. Therefore, devirtualization of these calls to methods of objects stored in maps requires an object-sensitive analysis that models abstract heap objects with context.

7 Cast Safety

We have used the points-to analysis results in a client analysis that proves that some casts cannot fail. A given cast cannot fail if the pointer that it is casting can only point to objects whose type is a subtype of the type of the cast. Table 8 shows the number of casts in each benchmark that cannot be statically proven safe by the cast safety analysis.

Context sensitivity improves precision of cast safety analysis in jack, javac, mpegaudio, mtrt, soot-c, sablecc-j, polyglot, antlr, bloat, chart, jython, pmd, and ps. Object sensitive cast safety analysis is never less precise and often significantly more precise than the call-site-string context sensitive variations. The improvements due to context sensitivity are most significant in the polyglot and javac benchmarks. In db, jack, javac, jess, soot-c, sablecc-j, polyglot, antlr, bloat, chart, jython, pmd, and ps, modelling abstract heap objects with receiver object context further improves precision of cast safety analysis.

The improvement is most dramatic in the polyglot benchmark, which contains a hierarchy of classes representing different kinds of nodes in an abstract syntax tree. At the root of this hierarchy is the Node_c class. This class implements a method called copy()

		object-sensitive				call site			
Benchmark	insens.	1	2	3	1H	1	2	1H	ZCWL
compress	**18**	18	18	18	18	18	18	18	18
db	27	27	27	27	**21**	27	27	27	27
jack	146	145	145	145	**104**	146	145	146	146
javac	405	370	370	370	**363**	391	370	391	
jess	130	130	130	130	**86**	130	130	130	130
mpegaudio	42	**38**	38	38	38	40	40	40	42
mtrt	31	**27**	27	27	27	27	27	27	29
soot-c	955	932	932	932	**878**	932	932	932	
sablecc-j	375	369	369	369	**331**	370	370	370	
polyglot	3539	3307	3306	3306	**1017**	3526	3443	3526	3318
antlr	295	275	275	275	**237**	276	275	276	276
bloat	1241	1207	1207		**1160**	1233	1207	1233	1234
chart	1097	1086	1085		**934**	1070	1070		
jython	501	499	499		**471**	499	499	499	499
pmd	1427	1376	1375	1375	**1300**	1393	1391	1393	
ps	641	612	612	612	**421**	612	612	612	612

Table 8. Number of casts potentially failing at run time

which, like the clone() method of Object, returns a copy of the node on which it is called. In fact, the copy() method first uses clone() to create the copy of the node, and then performs some additional processing on it. The static return type of copy() is Object, but at most sites calling it, the returned value is immediately cast to the static type of the node on which it is called. In our analysis, the clone() native method is modelled as returning its receiver; that is, the original object and the cloned version are represented by the same abstract object. Therefore, given a program that calls clone() directly, the cast safety analysis correctly determines that the run-time type of the clone is the same as that of the original. However, in polyglot, the call to clone() is wrapped inside copy(), and the casts appear at sites calling copy(). When copy() is analyzed in a context-insensitive way, it is deemed to possibly return any of the objects on which it is called throughout the program, so the casts cannot be proven to succeed. In an object-sensitive analysis, however, copy() is analyzed separately in the context of each receiver object on which it is called, and in each such context, it returns only an object of the same type as that receiver object. Therefore, the cast safety analysis proves statically that the casts of the return value of copy() cannot fail.

The number of potentially failing casts in the polyglot benchmark decreases even more dramatically between the 1-object-sensitive and 1H-object-sensitive columns of Table 8, from 3307 to 1017. The majority of these casts are in the parser generated by JavaCUP. The parser uses a Stack as the LR parse stack. Each object popped from the stack is cast to a Symbol. The generated polyglot parser contains about 2000 of these casts. The Stack class extends Vector, which uses an internal elementData array to store the objects that have been pushed onto the stack. In order to prove the safety of the casts, the analysis must distinguish the array implementing the parse stack from the arrays of other uses of Vector in the program. Since the array is allocated in one place, inside the Vector class, the different array instances can only be distinguished if abstract heap objects are modelled with context. Therefore, modelling abstract heap objects with object sensitivity is necessary to prove that these 2000 casts cannot fail.

8 Related Work

The most closely related work is the evaluation of object-sensitive analysis by Milanova, Rountev, and Ryder [17, 18]. They implemented a limited form of object sensitivity within their points-to analysis framework based on annotated constraints [21] and built on top of the BANE toolkit [2]. In particular, they selected a subset of pointer variables (method parameters, the this pointer, and the method return value) which they modelled context-sensitively using the receiver object as the context abstraction. All other pointer variables and all abstract heap objects were modelled without context. The precision of the analysis was evaluated on benchmarks using version 1.1.8 of the Java standard library, and compared to a context-insensitive and to a call-site context-sensitive analysis, using call graph construction, virtual call resolution, and mod-ref analysis as client analyses. Our BDD-based implementation has made it feasible to evaluate object-sensitive analysis on benchmarks using the much larger version 1.3.1_01 of the Java standard library. Thanks to the better scalability of the BDD-based implementation, we have performed a much broader empirical exploration of the design space of object-sensitive analyses. In particular, we have modelled all pointer variables context-sensitively, rather than only a subset, we have used receiver object strings of length up to three, rather than only one, and we have modelled abstract heap objects context-sensitively.

Whaley and Lam [29] suggest several client analyses of the ZCWL algorithm, but state that "in-depth analysis of the accuracy of the analyses . . . is beyond the scope of this paper." They do, however, provide some preliminary data about thread escape analysis and a "type refinement analysis" for finding variables whose declared type could be made more specific. In this paper, we have compared the precision of the ZCWL algorithm against object-sensitive and call-site-string context-sensitive analyses using several client analyses, namely call graph construction, virtual call resolution, and cast safety analysis.

Liang, Pennings and Harrold [16] evaluated the effect of context sensitivity on the size of pointed-to-by sets (the inverse of points-to sets), normalized using dynamic counts. Instead of using BDDs to allow their analyses to scale to benchmarks using the large Java standard library, they simulated the library with a hand-crafted model. Their results agree with our findings that context sensitivity improves precision for some benchmarks, and that a context-sensitive heap abstraction is important for precision. However, they found that call sites are sometimes more precise than receiver objects. This difference could be caused by several factors, including their different choice of benchmarks, their very different precision metric (pointed-to-by sets), or their simulation of the standard library.

Several context-sensitive points-to analyses other than the subset-based analyses studied in this paper have been proposed. Wilson and Lam [30] computed summary functions summarizing the effects of functions, which they then inlined into summaries of their callers. Liang and Harrold [15] proposed an equality-based context-sensitive analysis; its precision relative to subset-based context-sensitive analysis remains to be studied. Ruf [22] compared context-insensitive analysis to using "assumption sets" as the context abstraction, and concluded that on C benchmarks, context sensitivity had little effect on the points-to sets of pointers that are actually dereferenced. Like object sensitivity, the Cartesian Product Algorithm [1, 28] uses abstract objects as the context abstraction, but includes all method parameters as context, rather than only the receiver parameter. In the future, it would be interesting to empirically compare these additional variations of context-sensitive analysis with those studied in this paper.

9 Conclusions

We have performed an in-depth empirical study of the effects of variations of context sensitivity on the precision of Java points-to analysis. In particular, we studied object-sensitive analysis, context-sensitive analysis using call sites as the context abstraction, and the ZCWL algorithm. We evaluated the effects of these variations on the number of contexts generated, the number of distinct points-to sets constructed, and on the precision of call graph construction, virtual call resolution, and cast safety analysis.

Overall, we found that context sensitivity improved call graph precision by a small amount, improved the precision of virtual call resolution by a more significant amount, and enabled a major precision improvement in cast safety analysis.

Object-sensitive analysis was clearly better than the other variations of context sensitivity that we studied, both in terms of analysis precision and potential scalability. Client analyses based on object-sensitive analyses were never less precise than those based on call-site-string context-sensitive analyses or on the ZCWL algorithm, and in many cases, they were significantly more precise. As we increased the length of context strings, the number of abstract contexts produced with object-sensitive analysis grew much more slowly than with the other variations of context sensitivity, so object-sensitive analysis is likely to scale better. However, the number of equivalence classes of contexts was greater with object sensitivity than with the other variations, which indicates that object sensitivity better distinguishes contexts that give rise to differences in points-to sets.

Of the object-sensitive variations, extending the length of context strings caused very few additional improvements in analysis precision compared to 1-object-sensitive analysis. However, modelling abstract heap objects with context did improve precision significantly in many cases. Therefore, we conclude that 1-object-sensitive and 1H-object-sensitive analyses provide the best tradeoffs between precision and analysis efficiency. Our measurements of the numbers of abstract contexts and distinct points-to sets suggest that it should be feasible to implement an efficient non-BDD-based 1-object-sensitive analysis using current implementation techniques such as shared bit vectors. Efficiently implementing a 1H-object-sensitive analysis without BDDs will require new improvements in the data structures and algorithms used to implement points-to analyses, and we expect that our results will motivate and help guide this future research.

Although the ZCWL algorithm constructs call-site strings of arbitrary length, client analyses based on it were never more precise than those based on object-sensitive analysis. In many cases, analyses based on the ZCWL algorithm were even less precise than those based on 1-call-site-sensitive analysis. The key cause of the disappointing results of this algorithm was its context-insensitive treatment of calls within SCCs of the initial call graph — a large proportion of call edges were indeed within SCCs.

References

1. O. Agesen. The Cartesian product algorithm. In *ECOOP '95*, volume 952 of *LNCS*, pages 2–51, 1995.
2. A. Aiken, M. Faehndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Types in Compilation*, volume 1473 of *LNCS*, pages 78–96, 1998.
3. M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of PLDI 2003*, pages 103–114, 2003.
4. M. Burke, P. Carini, J. Choi, and M. Hind. Interprocedural pointer alias analysis. Technical Report RC 21055, IBM T. J. Watson Research Center, Dec. 1997.

5. DaCapo Benchmark Suite. `http://www-ali.cs.umass.edu/DaCapo/gcbm.html`.
6. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95*, volume 952 of *LNCS*, pages 77–101, 1995.
7. B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, June 2004.
8. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of PLDI 1994*, pages 242–256, 1994.
9. N. Heintze. Analysis of large code bases: the compile-link-analyze model. `http://cm.bell-labs.com/cm/cs/who/nch/cla.ps`, 1999.
10. M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of PASTE 2001*, pages 54–61. ACM Press, 2001.
11. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.
12. O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, Jan. 2006.
13. O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Apr. 2003.
14. O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *Proceedings of PLDI 2004*, pages 158–169. ACM Press, 2004.
15. D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *ESEC/FSE '99*, volume 1687 of *LNCS*, pages 199–215. Springer-Verlag / ACM Press, 1999.
16. D. Liang, M. Pennings, and M. J. Harrold. Evaluating the impact of context-sensitivity on andersen's algorithm for Java programs. In *PASTE 2005*. ACM Press, Sept. 2005.
17. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of ISSTA 2002*, pages 1–11. ACM Press, 2002.
18. 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.
19. E. M. Nystrom, H.-S. Kim, and W.-m. W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of PASTE 2004*, pages 43–48. ACM Press, 2004.
20. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction*, volume 2622 of *LNCS*, pages 138–152, 2003.
21. A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings OOPSLA 2001*, pages 43–55. ACM Press, 2001.
22. E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 13–22. ACM Press, 1995.
23. B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Compiler Construction*, volume 2622 of *LNCS*, pages 126–137. Springer, 2003.
24. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.
25. O. Shivers. Control flow analysis in scheme. In *Proceedings of PLDI 1988*, pages 164–174.
26. SPEC JVM98 Benchmarks. `http://www.spec.org/osg/jvm98/`.
27. R. Vallée-Rai. Ashes suite collection. `http://www.sable.mcgill.ca/ashes/`.
28. T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 99–117, 2001.
29. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of PLDI 2004*, pages 131–144. ACM Press, 2004.
30. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of PLDI 1995*, pages 1–12. ACM Press, 1995.
31. J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of PLDI 2004*, pages 145–157. ACM Press, 2004.