Context-Sensitive Points-to Analysis: Is It Worth It?

CC 2006

Ondřej Lhoták Laurie Hendren
Presented by Markus Kusano

September 15, 2015

Motivation

- Does context-sensitivity improve precision?

- The main goal of this study was to investigate if context-sensitivity improves the precision of inter-procedural analyses for object oriented programs
- As we've already seen, there are many different types of context-sensitivity
- This begs the question as to which type of context-sensitivity performs the best
- Finally, it would be interesting to know how many contexts an analysis produces
- The number of contexts may relate to both the precision and scalability of an analysis

Motivation

- Does context-sensitivity improve precision?
- Which type of context-sensitivity is the best?

- The main goal of this study was to investigate if context-sensitivity improves the precision of inter-procedural analyses for object oriented programs
- As we've already seen, there are many different types of context-sensitivity
- This begs the question as to which type of context-sensitivity performs the best
- Finally, it would be interesting to know how many contexts an analysis produces
- The number of contexts may relate to both the precision and scalability of an analysis

Motivation

- Does context-sensitivity improve precision?
- Which type of context-sensitivity is the best?
- How many contexts does an analysis produce?

- The main goal of this study was to investigate if context-sensitivity improves the precision of inter-procedural analyses for object oriented programs
- As we've already seen, there are many different types of context-sensitivity
- This begs the question as to which type of context-sensitivity performs the best
- Finally, it would be interesting to know how many contexts an analysis produces
- The number of contexts may relate to both the precision and scalability of an analysis

Contributions

- Java points-to comparison

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Contributions

- Java points-to comparison
- Comparison of four analyses:

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Contributions

- Java points-to comparison
- Comparison of four analyses:
  - Context insensitive

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Contributions

- Java points-to comparison
- Comparison of four analyses:
  - Context insensitive
  - Call-site sensitive

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Contributions

- Java points-to comparison
- Comparison of four analyses:
  - Context insensitive
  - Call-site sensitive
  - Object sensitive

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Contributions

- Java points-to comparison
- Comparison of four analyses:
  - Context insensitive
  - Call-site sensitive
  - Object sensitive
  - Acyclic Call-graph Paths (ZCWL)

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Contributions

- Java points-to comparison
- Comparison of four analyses:
  - Context insensitive
  - Call-site sensitive
  - Object sensitive
  - Acyclic Call-graph Paths (ZCWL)
- Quantitative comparison

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Contributions

- Java points-to comparison
- Comparison of four analyses:
  - Context insensitive
  - Call-site sensitive
  - Object sensitive
  - Acyclic Call-graph Paths (ZCWL)
- Quantitative comparison
  - Statistical summaries

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Contributions

- Java points-to comparison
- Comparison of four analyses:
  - Context insensitive
  - Call-site sensitive
  - Object sensitive
  - Acyclic Call-graph Paths (ZCWL)
- Quantitative comparison
  - Statistical summaries
- Qualitative comparison

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Contributions

- Java points-to comparison
- Comparison of four analyses:
  - Context insensitive
  - Call-site sensitive
  - Object sensitive
  - Acyclic Call-graph Paths (ZCWL)
- Quantitative comparison
  - Statistical summaries
- Qualitative comparison
  - Code patterns showing variation

- The authors contributions are answers to the previous questions
- Their comparison focuses on the effectiveness of different context-sensitivities for analyzing Java programs
- They implemented four different analyses within the same framework
- The first is a context insensitive analysis
- The second is a call-site sensitive algorithm using context strings
- The third is an object sensitive analysis
- And the fourth is a technique using the length of acyclic call-graph paths as the maxinum call-site abstraction size
- Their analysis is both qualitative and quantitative
- The qualitative results come from statistical summarizations of the effectiveness of the analysis
- They also show qualitative examples of types of code-patterns where the analyses show variations in effectiveness

Next, I'll provide a brief background on the different analyses the authors studied

Background: Abstractions

- Calling context

- Luckily, we've already looked at almost all the analyses the authors studied
- The authors investigate the effects of two different types of abstractions: calling context, and pointer allocation or heap abstractions
- I'll go over the high level details of both of these techniques

Background: Abstractions

- Calling context
- Pointer allocation (heap)

- Luckily, we've already looked at almost all the analyses the authors studied
- The authors investigate the effects of two different types of abstractions: calling context, and pointer allocation or heap abstractions
- I'll go over the high level details of both of these techniques

Background: Calling Context Abstraction

- Call-site context sensitivity

- We've seen presentations about different calling-context abstractions
- The first, call-site context sensitivity represents the calling context based on the location where the call was invoked
- In receiving-object context sensitivity, the context is based on the object on which the method is invoked
- In both of these cases, the context information is represented using bounded strings
- This is required to ensure termination because in general, the context information could be infinite, for example, if the program uses recursion
- The authors look at two different ways to bound the length of the context information
- The first is to use a fix bound k to limit the length
- The second is to use a bound from the longest path in the call-graph where strongly-connected components are merged.
- The authors refer to this second approach based on the creating authors names, ZCWL

Background: Calling Context Abstraction

- Call-site context sensitivity
- Receiving-object context sensitivity

- We've seen presentations about different calling-context abstractions
- The first, call-site context sensitivity represents the calling context based on the location where the call was invoked
- In receiving-object context sensitivity, the context is based on the object on which the method is invoked
- In both of these cases, the context information is represented using bounded strings
- This is required to ensure termination because in general, the context information could be infinite, for example, if the program uses recursion
- The authors look at two different ways to bound the length of the context information
- The first is to use a fix bound k to limit the length
- The second is to use a bound from the longest path in the call-graph where strongly-connected components are merged.
- The authors refer to this second approach based on the creating authors names, ZCWL

Background: Calling Context Abstraction

- Call-site context sensitivity
- Receiving-object context sensitivity
- Bounded by finite length strings

- We've seen presentations about different calling-context abstractions
- The first, call-site context sensitivity represents the calling context based on the location where the call was invoked
- In receiving-object context sensitivity, the context is based on the object on which the method is invoked
- In both of these cases, the context information is represented using bounded strings
- This is required to ensure termination because in general, the context information could be infinite, for example, if the program uses recursion
- The authors look at two different ways to bound the length of the context information
- The first is to use a fix bound k to limit the length
- The second is to use a bound from the longest path in the call-graph where strongly-connected components are merged.
- The authors refer to this second approach based on the creating authors names, ZCWL

Background: Calling Context Abstraction

- Call-site context sensitivity
- Receiving-object context sensitivity
- Bounded by finite length strings
  - Use fix bound k

- We've seen presentations about different calling-context abstractions
- The first, call-site context sensitivity represents the calling context based on the location where the call was invoked
- In receiving-object context sensitivity, the context is based on the object on which the method is invoked
- In both of these cases, the context information is represented using bounded strings
- This is required to ensure termination because in general, the context information could be infinite, for example, if the program uses recursion
- The authors look at two different ways to bound the length of the context information
- The first is to use a fix bound k to limit the length
- The second is to use a bound from the longest path in the call-graph where strongly-connected components are merged.
- The authors refer to this second approach based on the creating authors names, ZCWL

Background: Calling Context Abstraction

- Call-site context sensitivity
- Receiving-object context sensitivity
- Bounded by finite length strings
  - Use fix bound k
  - Longest non-cyclic path in the call-graph (ZCWL)

- We've seen presentations about different calling-context abstractions
- The first, call-site context sensitivity represents the calling context based on the location where the call was invoked
- In receiving-object context sensitivity, the context is based on the object on which the method is invoked
- In both of these cases, the context information is represented using bounded strings
- This is required to ensure termination because in general, the context information could be infinite, for example, if the program uses recursion
- The authors look at two different ways to bound the length of the context information
- The first is to use a fix bound k to limit the length
- The second is to use a bound from the longest path in the call-graph where strongly-connected components are merged.
- The authors refer to this second approach based on the creating authors names, ZCWL

Background: Calling Context Abstraction

```
...
A obj = new A();
...
```

- Context-insensitive: o_2

- An orthogonal decision is how to abstractly the object returned by an allocation operation
- In many of the previous analyses, we considered each allocation site to return one single abstract object
- Essentially, this meant we would create a point-to set for each object allocated on each line
- This approach considers the heap in a context-insensitive way
- Looking at this example we can see the creation of an object on line two. We can represent this allocation as an object o_2
- An alternative approach is to use either the calling-context or receiving-object context for pointer allocations
- For example, suppose this allocation occurs in calling context c_1.
- We can instead represent the allocation as the pair (c_1, o_2)
- In this way, we treat each allocation in every context as distinct
- A similar abstraction can be done using the receiving object

Background: Calling Context Abstraction

```
1    ...
2    A obj = new A();
3    ...
```

- Context-insensitive: o_2
- Calling-context c_1

- An orthogonal decision is how to abstractly the object returned by an allocation operation
- In many of the previous analyses, we considered each allocation site to return one single abstract object
- Essentially, this meant we would create a point-to set for each object allocated on each line
- This approach considers the heap in a context-insensitive way
- Looking at this example we can see the creation of an object on line two. We can represent this allocation as an object o_2
- An alternative approach is to use either the calling-context or receiving-object context for pointer allocations
- For example, suppose this allocation occurs in calling context c_1.
- We can instead represent the allocation as the pair (c_1, o_2)
- In this way, we treat each allocation in every context as distinct
- A similar abstraction can be done using the receiving object

Background: Calling Context Abstraction

```
1    ...
2    A obj = new A();
3    ...
```

- Context-insensitive: o_2
- Calling-context c_1
- Calling-context Heap Abstraction: (c_1, o_2)

- An orthogonal decision is how to abstractly the object returned by an allocation operation
- In many of the previous analyses, we considered each allocation site to return one single abstract object
- Essentially, this meant we would create a point-to set for each object allocated on each line
- This approach considers the heap in a context-insensitive way
- Looking at this example we can see the creation of an object on line two. We can represent this allocation as an object o_2
- An alternative approach is to use either the calling-context or receiving-object context for pointer allocations
- For example, suppose this allocation occurs in calling context c_1.
- We can instead represent the allocation as the pair (c_1, o_2)
- In this way, we treat each allocation in every context as distinct
- A similar abstraction can be done using the receiving object

Benchmarks

Benchmark	Total number of classes	Total number of methods	Executed methods app.	Executed methods +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

- The authors performed their analysis on a set of programs from a variety of different benchmark suites
- Their analysis included all application and library code except for the Java standard library
- On the far left we can see the total number of classes and methods
- The authors also then executed the benchmarks and counted the number of methods executed
- The left column labeled "app" shows the number of methods executed excluding the Java standard library
- The far right column shows the number of methods including the standard library

Next, we'll start looking at the results starting with the number of contexts produced by the different abstractions

Counting Contexts

- Count method–context pairs

- To count the number of contexts, the authors consider pairs of methods and calling contexts as a single "context"
- For example, consider the object-sensitive abstraction
- If we have some object abstraction o and two of its methods m_1 and m_2, the authors count the invocations of m_1 and m_2 with the same object abstraction as a single context

Counting Contexts

- Count method–context pairs
- Object-abstraction o and methods m_1 and m_2

- To count the number of contexts, the authors consider pairs of methods and calling contexts as a single "context"
- For example, consider the object-sensitive abstraction
- If we have some object abstraction o and two of its methods m_1 and m_2, the authors count the invocations of m_1 and m_2 with the same object abstraction as a single context

Counting Contexts

- Count method–context pairs
- Object-abstraction o and methods m_1 and m_2
- Two contexts: (m_1, o), and (m_2, o)

- To count the number of contexts, the authors consider pairs of methods and calling contexts as a single "context"
- For example, consider the object-sensitive abstraction
- If we have some object abstraction o and two of its methods m_1 and m_2, the authors count the invocations of m_1 and m_2 with the same object abstraction as a single context

Context Sizes

- Bounded size of context information (1,2,3)
- ZCWL bound
- 1H: size one bound for calling-context and heap abstraction

- As we'll see in a second, the authors used varying bound sizes for each analysis
- The bound size is simply represented as an integer
- They also used the ZCWL bound computed using the call-graph
- The call-graph used was created from the context-insensitive analysis
- Finally, when the authors say "1H" they use a size one calling-context and a size-one heap abstraction

Number of Contexts

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

- This table shows the results comparing the number of contexts for the different abstractions
- On the far right, "insen" shows the number of "contexts" for the context-insensitive analysis
- Since the context-insensitive analysis, conceptually, has a single context for each method invocation, this column is simply the number of method invocations
- The values in the columns to the right are all showing the number of contexts as a multiple of insens
- For example, the 1 object sensitive analysis has 13.7 times the number of contexts as the insen analysis
- Columns which are blank indicate the system ran out of memory
- The results show that there is a very large increase in memory as the amount of context information increases
- This means that explicitly representing the context information for large programs will not scale

Next, we'll look at results investigating the number of equivalent contexts

Equivalent Contexts

- Method–context pairs: (m_1, c_1), (m_2, c_2)

- The authors further examined all the method–context pairs to investigate which of the pairs was equivalent
- They define equivalence of two pairs with methods m_1 and m_2 and context c_1 and c_2 to require that the two methods are the same and for all pointer variables in the method, the points-to set of the point is the same in both contexts
- In essence, this notion of equivalence means that if two pairs are equivalent, we would have been better off only keeping one of the contexts to save memory

Equivalent Contexts

- Method–context pairs: (m_1, c_1), (m_2, c_2)
- Two method–context pairs are equivalent if:

- The authors further examined all the method–context pairs to investigate which of the pairs was equivalent
- They define equivalence of two pairs with methods m_1 and m_2 and context c_1 and c_2 to require that the two methods are the same and for all pointer variables in the method, the points-to set of the point is the same in both contexts
- In essence, this notion of equivalence means that if two pairs are equivalent, we would have been better off only keeping one of the contexts to save memory

Equivalent Contexts

- Method–context pairs: (m_1, c_1), (m_2, c_2)
- Two method–context pairs are equivalent if:
  - $m_1 = m_2$

- The authors further examined all the method–context pairs to investigate which of the pairs was equivalent
- They define equivalence of two pairs with methods m_1 and m_2 and context c_1 and c_2 to require that the two methods are the same and for all pointer variables in the method, the points-to set of the point is the same in both contexts
- In essence, this notion of equivalence means that if two pairs are equivalent, we would have been better off only keeping one of the contexts to save memory

Equivalent Contexts

- Method–context pairs: (m_1, c_1), (m_2, c_2)
- Two method–context pairs are equivalent if:
  - $m_1 = m_2$
  - For all pointer variables p in m_1, the points-to set of p is the same in c_1 and c_2

- The authors further examined all the method–context pairs to investigate which of the pairs was equivalent
- They define equivalence of two pairs with methods m_1 and m_2 and context c_1 and c_2 to require that the two methods are the same and for all pointer variables in the method, the points-to set of the point is the same in both contexts
- In essence, this notion of equivalence means that if two pairs are equivalent, we would have been better off only keeping one of the contexts to save memory

Equivalent Contexts

- Method–context pairs: (m_1, c_1), (m_2, c_2)
- Two method–context pairs are equivalent if:
  - $m_1 = m_2$
  - For all pointer variables p in m_1, the points-to set of p is the same in c_1 and c_2
- Equivalent pairs means context information does not provide extra information

- The authors further examined all the method–context pairs to investigate which of the pairs was equivalent
- They define equivalence of two pairs with methods m_1 and m_2 and context c_1 and c_2 to require that the two methods are the same and for all pointer variables in the method, the points-to set of the point is the same in both contexts
- In essence, this notion of equivalence means that if two pairs are equivalent, we would have been better off only keeping one of the contexts to save memory

Equivalent Contexts

Benchmark	insens.	object-sensitive 1	object-sensitive 2	object-sensitive 3	object-sensitive 1H	call site 1	call site 2	call site 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

- This table shows the number of equivalence classes for all the techniques examined
- Again, the number of equivalence classes shows how beneficial the extra context information was: less equivalence classes means the context-information did not provide extra precision
- Again, the columns of all the context-sensitive analyses are multiples of the insens column.
- Here, we can see the object sensitive analysis creates more equivalance classes, or, that it is able to partition the results into more non-equivalant groups
- This means the object-sensitive abstraction may be better at providing extra precision using the context information compared to the call-site abstraction
- Also, we can see the number of equivalence classes does not increase too much as the size of the context increases
- This means that the analysis results do not improve too much with larger contexts
- Interestingly, we see that ZCWL preforms rather poorly since the effective context-size bound used by the analysis is always much larger than 2
- However, the ZCWL method merges call-graph nodes in strongly-connected components and treats them in a context-insensitive manner
- The authors found that a large porition of the call-graph of many of the bench marks is a strongly-connected component resulting in the ZCWL method to degrade to a context

Next, we'll look at the applicability of the different context-sensitivities in performing client analyses

Reachable Methods

Benchmark	CHA	insens.	object-sensitive 1	object-sensitive 2	object-sensitive 3	object-sensitive 1H	call site 1	call site 2	call site 1H	actually 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

- This table shows the number of reachable methods created by the object and call-site sensitive analyses
- For each benchmark, the most precise and least-expensive analysis has been highlighted in bold
- Overall, we can see the points-to based approach can significantly improve over CHA
- The 1-object-sensitive analysis can slightly improve over the insensitive analysis
- The call-site sensitive analysis can approach the performance of the object-sensitive analysis but often requires larger context information

Potentially Polymorphic Functions

			object-sensitive				call site		
Benchmark	CHA	insens.	1	2	3	1H	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

- This table shows the number of potentially polymorphic functions in the call-graph of the different analyses
- In other words, these are all the call-sites with more than one outgoing edge
- The authors notes that the benchmarks which are written in a more object-oriented style can be better handled by the object-sensitive analysis compared to the insensitive analysis
- The call-site context analysis can sometimes match the performance of the object-sensitive analysis but never is more acurrate

Cast Safety

Benchmark	insens.	object-sensitive				call site			ZCWL
		1	2	3	1H	1	2	1H	
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

- The authors created a cast-safety analysis which deems a runtime cast as safe if the pointer being casted could only point to subtypes of the casted type, otherwise, the cast may be unsafe
- The table shows the number of potentially unsafe casts for each analysis
- The cast-safety results are similar to the results of the previous analyses
- The object-sensitive analysis is never less precise than the call-site sensitive analysis and is often significantly more precise

Conclusion

- Comparison of various types of context-sensitivity on scalability and precision

Conclusion

- Comparison of various types of context-sensitivity on scalability and precision
- Showed effects of context sensitivity on many client analyses

Conclusion

- Comparison of various types of context-sensitivity on scalability and precision
- Showed effects of context sensitivity on many client analyses
- Analyzing Java? Wanna use some kind of sensitivity?

Conclusion

- Comparison of various types of context-sensitivity on scalability and precision
- Showed effects of context sensitivity on many client analyses
- Analyzing Java? Wanna use some kind of sensitivity?
  - Use an object-sensitive analysis