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
• 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 maximum 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 maximum 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 maximum 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 maximum 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 maximum 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 maximum 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
• 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 maximum 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 maximum 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
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 maximum 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 maximum 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
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.
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.
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.
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

1
A obj = new A();

• 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

▶ 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.

---

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

- Context-insensitive: $o_2$
- Calling-context $c_1$
Background: Calling Context Abstraction

- 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

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

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total number of classes</th>
<th>Executed methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>methods</td>
<td>app</td>
</tr>
<tr>
<td>compress</td>
<td>41</td>
<td>56</td>
</tr>
<tr>
<td>db</td>
<td>32</td>
<td>51</td>
</tr>
<tr>
<td>jack</td>
<td>86</td>
<td>291</td>
</tr>
<tr>
<td>javac</td>
<td>209</td>
<td>778</td>
</tr>
<tr>
<td>jess</td>
<td>180</td>
<td>395</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>88</td>
<td>222</td>
</tr>
<tr>
<td>mrtt</td>
<td>55</td>
<td>182</td>
</tr>
<tr>
<td>soot-c</td>
<td>731</td>
<td>1055</td>
</tr>
<tr>
<td>sablesc-p</td>
<td>342</td>
<td>1034</td>
</tr>
<tr>
<td>polyglot</td>
<td>502</td>
<td>2037</td>
</tr>
<tr>
<td>antlr</td>
<td>203</td>
<td>1099</td>
</tr>
<tr>
<td>bloat</td>
<td>434</td>
<td>138</td>
</tr>
<tr>
<td>chart</td>
<td>1077</td>
<td>854</td>
</tr>
<tr>
<td>jython</td>
<td>270</td>
<td>1004</td>
</tr>
<tr>
<td>pmd</td>
<td>1546</td>
<td>1817</td>
</tr>
<tr>
<td>ps</td>
<td>202</td>
<td>285</td>
</tr>
</tbody>
</table>
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
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 insen.

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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>insens.</th>
<th>object-sensitive</th>
<th>call site 1</th>
<th>call site 2</th>
<th>call site 1H</th>
<th>call site 2H</th>
<th>ZCWL</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>2596</td>
<td>13.7 113 1517</td>
<td>13.4</td>
<td>6.5</td>
<td>237 6.5</td>
<td>2.9 × 10⁴</td>
<td></td>
</tr>
<tr>
<td>db</td>
<td>2613</td>
<td>13.7 115 1555</td>
<td>13.4</td>
<td>6.5</td>
<td>236 6.5</td>
<td>7.9 × 10⁴</td>
<td></td>
</tr>
<tr>
<td>jack</td>
<td>2869</td>
<td>13.8 156 1872</td>
<td>13.2</td>
<td>6.8</td>
<td>220 6.8</td>
<td>2.7 × 10⁴</td>
<td></td>
</tr>
<tr>
<td>javac</td>
<td>3780</td>
<td>15.8 297 13289</td>
<td>15.6</td>
<td>8.4</td>
<td>244 8.4</td>
<td>10.4 × 10⁴</td>
<td></td>
</tr>
<tr>
<td>jess</td>
<td>3216</td>
<td>19.0 305 5394</td>
<td>18.6</td>
<td>6.7</td>
<td>207 6.7</td>
<td>6.1 × 10⁵</td>
<td></td>
</tr>
<tr>
<td>mpedeaudoo</td>
<td>2793</td>
<td>13.0 107 1419</td>
<td>12.7</td>
<td>6.3</td>
<td>221 6.3</td>
<td>4.4 × 10⁵</td>
<td></td>
</tr>
<tr>
<td>mitl</td>
<td>2738</td>
<td>13.3 108 1447</td>
<td>13.1</td>
<td>6.6</td>
<td>226 6.6</td>
<td>1.2 × 10⁶</td>
<td></td>
</tr>
<tr>
<td>soot-c</td>
<td>4837</td>
<td>11.1 168 4010</td>
<td>10.9</td>
<td>8.2</td>
<td>198 8.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sabaci-cj</td>
<td>5608</td>
<td>10.8 116 1792</td>
<td>10.5</td>
<td>5.5</td>
<td>126 5.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>polyglot</td>
<td>5616</td>
<td>11.7 149 2091</td>
<td>11.2</td>
<td>7.1</td>
<td>144 7.1</td>
<td>10130</td>
<td></td>
</tr>
<tr>
<td>antrf</td>
<td>3897</td>
<td>15.0 309 8110</td>
<td>14.7</td>
<td>9.6</td>
<td>191 9.6</td>
<td>4.8 × 10⁷</td>
<td></td>
</tr>
<tr>
<td>bloat</td>
<td>5237</td>
<td>14.3 291 140</td>
<td>14.0</td>
<td>8.9</td>
<td>159 8.9</td>
<td>3.0 × 10⁸</td>
<td></td>
</tr>
<tr>
<td>chart</td>
<td>7069</td>
<td>22.3 500 219</td>
<td>21.9</td>
<td>7.0</td>
<td>335</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>4401</td>
<td>18.8 384 183</td>
<td>18.3</td>
<td>6.7</td>
<td>162 6.7</td>
<td>2.1 × 10¹⁵</td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>7219</td>
<td>13.4 283 5607</td>
<td>12.9</td>
<td>6.6</td>
<td>289 6.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ps</td>
<td>3874</td>
<td>13.3 271 24967</td>
<td>13.1</td>
<td>9.0</td>
<td>224 9.0</td>
<td>2.0 × 10⁸</td>
<td></td>
</tr>
</tbody>
</table>
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:
  - \(m_1 = m_2\)
  - 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 \(c_2\)

- 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
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 equivalence classes, or, that it is able to partition the results into more non-equivalent 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 portion of the call-graph of many of the benchmarks is a strongly-connected component resulting in the ZCWL method to degrade to a context-insensitive one.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>insens</th>
<th>object-sensitive</th>
<th>call site</th>
<th>ZCWL</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>2597</td>
<td>8.4 9.9 11.3 12.1</td>
<td>2.4 3.9 4.9</td>
<td>3.3</td>
</tr>
<tr>
<td>db</td>
<td>2614</td>
<td>8.5 9.9 11.4 12.1</td>
<td>2.4 3.9 5.0</td>
<td>3.3</td>
</tr>
<tr>
<td>jack</td>
<td>2870</td>
<td>8.6 10.2 11.6 11.9</td>
<td>2.4 3.9 5.0</td>
<td>3.4</td>
</tr>
<tr>
<td>javac</td>
<td>3781</td>
<td>10.4 17.7 33.8 14.3</td>
<td>2.7 5.3 5.4</td>
<td></td>
</tr>
<tr>
<td>jess</td>
<td>3217</td>
<td>8.9 10.6 12.0 13.9</td>
<td>2.6 4.2 5.0</td>
<td>3.9</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>2794</td>
<td>8.1 9.4 10.8 11.5</td>
<td>2.4 3.8 4.8</td>
<td>3.3</td>
</tr>
<tr>
<td>mrt</td>
<td>2793</td>
<td>8.3 9.7 11.1 11.8</td>
<td>2.5 4.0 4.9</td>
<td>3.4</td>
</tr>
<tr>
<td>soot-c</td>
<td>4838</td>
<td>7.1 13.7 18.4 9.8</td>
<td>2.6 4.2 4.8</td>
<td></td>
</tr>
<tr>
<td>sabie-cj</td>
<td>5609</td>
<td>6.9 8.4 9.6 9.5</td>
<td>2.3 3.6 3.9</td>
<td></td>
</tr>
<tr>
<td>polyglot</td>
<td>5617</td>
<td>7.9 9.4 10.8 10.2</td>
<td>2.4 3.7 4.7</td>
<td>3.3</td>
</tr>
<tr>
<td>antlr</td>
<td>3898</td>
<td>9.4 12.1 13.8 13.2</td>
<td>2.5 4.1 5.2</td>
<td>4.3</td>
</tr>
<tr>
<td>bloat</td>
<td>5238</td>
<td>10.2 44.6 12.9</td>
<td>2.8 4.9 5.2</td>
<td>6.7</td>
</tr>
<tr>
<td>chart</td>
<td>7070</td>
<td>10.0 17.4 18.2</td>
<td>2.7 4.8 5.4</td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>4402</td>
<td>9.9 55.9 15.6</td>
<td>2.5 4.3 4.6</td>
<td>4.0</td>
</tr>
<tr>
<td>pmd</td>
<td>7220</td>
<td>7.6 14.6 17.0 11.0</td>
<td>2.4 4.2 4.2</td>
<td></td>
</tr>
<tr>
<td>ps</td>
<td>3875</td>
<td>8.7 9.9 11.0 12.0</td>
<td>2.6 4.0 5.2</td>
<td>4.4</td>
</tr>
</tbody>
</table>

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 portion of the call-graph of many of the benchmarks is a strongly-connected component resulting in the ZCWL method to degrade to a context-insensitive one.
Next, we'll look at the applicability of the different context-sensitivities in performing client analyses.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CHA</th>
<th>insens.</th>
<th>object-sensitive</th>
<th>call site</th>
<th>actually executed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1    2   3  1H</td>
<td>1   2  3 1H</td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>90</td>
<td>59</td>
<td>59   59  59 59</td>
<td>59 59 59</td>
<td>56</td>
</tr>
<tr>
<td>db</td>
<td>95</td>
<td>65</td>
<td>64   64  64 64</td>
<td>65 64 65</td>
<td>51</td>
</tr>
<tr>
<td>jack</td>
<td>348</td>
<td>317</td>
<td>313  313 313 313</td>
<td>316 313 316</td>
<td>291</td>
</tr>
<tr>
<td>javac</td>
<td>1185</td>
<td>1154</td>
<td>1147 1147 1147 1147</td>
<td>1147 1147 1147</td>
<td>779</td>
</tr>
<tr>
<td>jess</td>
<td>683</td>
<td>630</td>
<td>629  629 629 623</td>
<td>629 629 629</td>
<td>395</td>
</tr>
<tr>
<td>megapack</td>
<td>306</td>
<td>255</td>
<td>251  251 251 251</td>
<td>251 251 251</td>
<td>222</td>
</tr>
<tr>
<td>mnt</td>
<td>217</td>
<td>189</td>
<td>186  186 186 186</td>
<td>187 187 187</td>
<td>182</td>
</tr>
<tr>
<td>soot-c</td>
<td>2395</td>
<td>2273</td>
<td>2264 2264 2264 2264</td>
<td>2266 2266 2266</td>
<td>1055</td>
</tr>
<tr>
<td>sablonce-j</td>
<td>1904</td>
<td>1744</td>
<td>1744 1744 1744 1731</td>
<td>1744 1744 1744</td>
<td>1034</td>
</tr>
<tr>
<td>polyglot</td>
<td>2540</td>
<td>2421</td>
<td>2419 2419 2419 2419</td>
<td>2419 2419 2419</td>
<td>2037</td>
</tr>
<tr>
<td>anfr</td>
<td>1374</td>
<td>1323</td>
<td>1323 1323 1323 1323</td>
<td>1323 1323 1323</td>
<td>1099</td>
</tr>
<tr>
<td>biot</td>
<td>2879</td>
<td>2464</td>
<td>2451 2451 2451 2451</td>
<td>2451 2451 2451</td>
<td>138</td>
</tr>
<tr>
<td>chart</td>
<td>3227</td>
<td>2081</td>
<td>2080 2080 2080 2031</td>
<td>2080 2080 2080</td>
<td>854</td>
</tr>
<tr>
<td>jython</td>
<td>2007</td>
<td>1695</td>
<td>1693 1693 1693 1683</td>
<td>1694 1693 1694</td>
<td>1064</td>
</tr>
<tr>
<td>mpm</td>
<td>4997</td>
<td>4528</td>
<td>4521 4521 4521 4509</td>
<td>4521 4521 4521</td>
<td>1817</td>
</tr>
<tr>
<td>ps</td>
<td>840</td>
<td>835</td>
<td>835  835 835 834</td>
<td>835 835 835</td>
<td>285</td>
</tr>
</tbody>
</table>

- 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

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 note 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 accurate.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CHA</th>
<th>insens.</th>
<th>object-sensitive</th>
<th>call site</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>16</td>
<td>3</td>
<td>3 3 3 3 3</td>
<td>3 3 3 3</td>
</tr>
<tr>
<td>db</td>
<td>36</td>
<td>5</td>
<td>4 4 4 4 4</td>
<td>5 5 5 5</td>
</tr>
<tr>
<td>jack</td>
<td>474</td>
<td>25</td>
<td>23 23 23 22</td>
<td>24 24 24</td>
</tr>
<tr>
<td>javac</td>
<td>908</td>
<td>73</td>
<td>720 720 720 720</td>
<td>720 720 720</td>
</tr>
<tr>
<td>jess</td>
<td>121</td>
<td>45</td>
<td>45 45 45 45</td>
<td>45 45 45</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>40</td>
<td>27</td>
<td>24 24 24 24</td>
<td>24 24 24</td>
</tr>
<tr>
<td>mrtt</td>
<td>20</td>
<td>9</td>
<td>7 7 7 7</td>
<td>8 8 8 8</td>
</tr>
<tr>
<td>soolt-c</td>
<td>1748</td>
<td>983</td>
<td>913 913 913 913</td>
<td>938 913 938</td>
</tr>
<tr>
<td>sablecc-s</td>
<td>722</td>
<td>450</td>
<td>325 325 325 325</td>
<td>380 325 380</td>
</tr>
<tr>
<td>polyglot</td>
<td>1332</td>
<td>744</td>
<td>592 592 592 585</td>
<td>592 592 592</td>
</tr>
<tr>
<td>antlr</td>
<td>1096</td>
<td>843</td>
<td>843 843 843 843</td>
<td>843 843 843</td>
</tr>
<tr>
<td>bloat</td>
<td>2503</td>
<td>1079</td>
<td>962 962 961</td>
<td>962 962 962</td>
</tr>
<tr>
<td>chart</td>
<td>2782</td>
<td>254</td>
<td>235 235 214</td>
<td>235 235 214</td>
</tr>
<tr>
<td>jython</td>
<td>646</td>
<td>347</td>
<td>347 347 346</td>
<td>347 347 347</td>
</tr>
<tr>
<td>pmd</td>
<td>2608</td>
<td>1224</td>
<td>1193 1193 1193 1163</td>
<td>1205 1205 1205</td>
</tr>
<tr>
<td>ps</td>
<td>321</td>
<td>304</td>
<td>303 303 303 300</td>
<td>303 303 303</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>insens.</th>
<th>object-sensitive</th>
<th>call site</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3</td>
<td>1 2</td>
<td>1 2</td>
</tr>
<tr>
<td>compress db</td>
<td>18</td>
<td>18 18 18 18 18</td>
<td>18 18</td>
</tr>
<tr>
<td>jack</td>
<td>146</td>
<td>145 145 145 145</td>
<td>146 145</td>
</tr>
<tr>
<td>javac</td>
<td>405</td>
<td>370 370 370 363</td>
<td>391 370</td>
</tr>
<tr>
<td>jess</td>
<td>130</td>
<td>130 130 130 86</td>
<td>130 130</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>42</td>
<td>38 38 38 38</td>
<td>40 40 40</td>
</tr>
<tr>
<td>mrt</td>
<td>31</td>
<td>27 27 27 27</td>
<td>27 27 27</td>
</tr>
<tr>
<td>soot-c</td>
<td>955</td>
<td>932 932 932 878</td>
<td>932 932</td>
</tr>
<tr>
<td>sablecc-j</td>
<td>375</td>
<td>369 369 369 331</td>
<td>370 370</td>
</tr>
<tr>
<td>polyglot</td>
<td>3539</td>
<td>3307 3306 3306 1017</td>
<td>3526 3443 3526</td>
</tr>
<tr>
<td>anfr</td>
<td>295</td>
<td>275 275 275 237</td>
<td>276 275</td>
</tr>
<tr>
<td>bloat</td>
<td>1241</td>
<td>1207 1207 1160</td>
<td>1233 1207</td>
</tr>
<tr>
<td>chart</td>
<td>1097</td>
<td>1086 1085 934</td>
<td>1070 1070</td>
</tr>
<tr>
<td>jython</td>
<td>501</td>
<td>499 499 499 471</td>
<td>499 499</td>
</tr>
<tr>
<td>pmr</td>
<td>1427</td>
<td>1376 1375 1375 1300</td>
<td>1393 1391 1393</td>
</tr>
<tr>
<td>ptd</td>
<td>641</td>
<td>612 612 612 612</td>
<td>612 612</td>
</tr>
</tbody>
</table>
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