“Context-sensitive Points-to Analysis: Is it worth it?”

by Ondřej Lhoták and Laurie Hendren

Presented by Nicholas Rudzicz

Question

- Is context-sensitive points-to analysis worth it?

Question

- Is context-sensitive points-to analysis worth it?
  - Yes.

The End

~~The End~~

Outline

- Course refresher
  - Motivation
- Background
  - Pedantic stuff
- for(int i=0; i < num_benchmarks; i++)
  - { describe_benchmark(i);
  - discuss_results_of_benchmark(i); }
- Conclusions

Course refresher

- Recall this example (without pointers) from class:

```
main() {
    g1=1;
    f(); //call_1
    ...
    g1=2;
    f(); //call_2
}

f() {
    g(); //call_3
}

g() {
    g1 = g1+1;
}
```

Course refresher

- Recall this example (without pointers) from class:

```
main() {
    g1=1;
    f();  //call_1
    ...
    g1=2;
    f();  //call_2
}

f() {
    g();   //call_3
}

g() {
    g1 = g1+1;
}
```

g1 = ?

g1 = ?

- Interprocedural analysis
  - W/out context, know nothing about g1 after calls

Course refresher

- Recall this example (without pointers) from class:

```
main() {
    g1=1;
    f();  //call_1
    ...
    g1=2;
    f();  //call_2
}

f() {
    g();  //call_3
}

g() {
    g1 = g1+1;
}
```

g1 = 2 (after call_1)

g1 = 3 (after call_2)

- Interprocedural analysis
  - W/out context, know nothing about g1 after calls
  - *With* context, we can make statements about g1

Course refresher

- Recall this example (without pointers) from class:

```
main() {
    g1=1;
    f();  //call_1
    ...
    g1=2;
    f();  //call_2
}

f() {
    g();   //call_3
}

g() {
    g1 = g1+1;
}
```

g1 = 2

g1 = 3

- Interprocedural analysis
  - W/out context, know nothing about g1 after calls
  - *With* context, we can make statements about g1 *at the cost of exponential code blowup or possibly infinite context strings*

Points-to Analysis

- Similar problem with pointers
  - What *could* p1 point to after a given call f()?
  - {(pp→p), (p→x)?, (p→y)?, (q→z)}

Points-to Analysis

- Similar problem with pointers
  - What *could* p1 point to after a given call f()?
  - {(pp→p), (p→x)?, (p→y)?, (q→z)}
- We would like to perform context-sensitive analysis, without the exponential blowup
  - Binary Decision Diagrams (BDDs) provide an efficient implementation, which we've seen

Points-to Analysis

- Similar problem with pointers
  - What *could* p1 point to after a given call f()?
  - {(pp→p), (p→x)?, (p→y)?, (q→z)}
- We would like to perform context-sensitive analysis, without the exponential blowup
  - Binary Decision Diagrams (BDDs) provide an efficient implementation, which we've seen
- Efficient experimentation is now possible
  - Forms the basis of this paper

L&H Paper: Background

- Examines the results of context-sensitive analysis

L&H Paper: Background

- Examines the results of context-sensitive analysis
- Pointer and pointer target abstractions are fixed
  - Store local variable and allocation statement, respectively

L&H Paper: Background

- Examines the results of context-sensitive analysis
- Pointer and pointer target abstractions are fixed
  - Store local variable and allocation statement, respectively
- Experiment with context abstraction models

L&H Paper: Background

- Examines the results of context-sensitive analysis
- Pointer and pointer target abstractions are fixed
  - Store local variable and allocation statement, respectively
- Experiment with context abstraction models
  - Call-site sensitivity
  - Receiver object sensitivity
  - Zhu & Calman, Whaley & Lam (ZCWL) algorithm

L&H Paper: Background

- Examines the results of context-sensitive analysis
- Pointer and pointer target abstractions are fixed
  - Store local variable and allocation statement, respectively
- Experiment with context abstraction models
  - Call-site sensitivity
  - Receiver object sensitivity
    - *1, 2, and 3-level context strings*
    - *1H – context-sensitive heap*
  - Zhu & Calman, Whaley & Lam (ZCWL) algorithm
    - Call-site abstraction. No bound on length of context string, but removes all cycles in context-insensitive graph to guarantee context string is finite.

Benchmarks

Benchmark	Total number of classes	methods	Executed 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

** from [1]*

- Tests performed on several benchmark suites
  - SpecJVM 98, DaCapo v.beta050224, Ashes

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

** from [1]*

- Tests performed on several benchmark suites
  - SpecJVM 98, DaCapo v.beta050224, Ashes
- Context-insensitive baseline tested first
- All variations of object-sensitive, call-site, and ZCWL-based analyses compared against this reference

Number of Contexts

- Simply a count of the number of contexts generated

Number of Contexts

- Simply a count of the number of contexts generated
- Context-insensitive (CI) versions ~2500-7000 contexts
  - Equal to the number of methods, since each one has a single "context"

Number of Contexts

- Simply a count of the number of contexts generated
- Context-insensitive (CI) versions ~2500-7000 contexts
  - Equal to the number of methods, since each one has a single "context"
- 1-level object-sensitive (OS) and call site (CS) contexts generate roughly 10-20 and 5-10 times the CI contexts
  - 1H sensitivity gives approximately the same numbers

Number of Contexts

- Simply a count of the number of contexts generated
- Context-insensitive (CI) versions ~2500-7000 contexts
  - Equal to the number of methods, since each one has a single "context"
- 1-level object-sensitive (OS) and call site (CS) contexts generate roughly 10-20 and 5-10 times the CI contexts
  - 1H sensitivity gives approximately the same numbers
- 2-level sensitivity generates ~100-500 and ~125-350 times the CI contexts (3-level OS generates ~1500-25,000 times)

Number of Contexts

- Simply a count of the number of contexts generated
- Context-insensitive (CI) versions ~2500-7000 contexts
  - Equal to the number of methods, since each one has a single "context"
- 1-level object-sensitive (OS) and call site (CS) contexts generate roughly 10-20 and 5-10 times the CI contexts
  - 1H sensitivity gives approximately the same numbers
- 2-level sensitivity generates ~100-500 and ~125-350 times the CI contexts (3-level OS generates ~1500-25,000 times)
- ZCWL generates between 2.9×10^4 and 2.1×10^{15} times the contexts!

Number of Contexts

- Huge numbers of contexts
  - Explicitly representing each one is a recipe for disaster

Number of Contexts

- Huge numbers of contexts
  - Explicitly representing each one is a recipe for disaster
- Explains why previous analyses could not scale to the benchmarks used in this case study

Equivalent Contexts

- Many contexts can be considered equivalent

Equivalent Contexts

- Many contexts can be considered equivalent
- Formally:
  - Two method-context pairs (m_1, c_1) and (m_2, c_2) are *equivalent* if $m_1 = m_2$, and any local pointer *p* has the same points-to set in both contexts

Equivalent Contexts

- Many contexts can be considered equivalent
- Formally:
  - Two method-context pairs (m_1, c_1) and (m_2, c_2) are *equivalent* if $m_1 = m_2$, and any local pointer *p* has the same points-to set in both contexts
- If there are many equivalent contexts in an analysis, explicitly storing each one separately is a waste

Equivalent Contexts

- Many contexts can be considered equivalent
- Formally:
  - Two method-context pairs (m_1, c_1) and (m_2, c_2) are *equivalent* if $m_1 = m_2$, and any local pointer *p* has the same points-to set in both contexts
- If there are many equivalent contexts in an analysis, explicitly storing each one separately is a waste
- However, methods to determine equivalent contexts prior to analysis have yet to be discovered

Equivalent Contexts

- In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted

Equivalent Contexts

- In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted
- 1-level OS- and CS-based analyses dropped to only ~8-10 and 2-3 times the CI contexts
  - From 10-20 and 5-10 times, previously

Equivalent Contexts

- In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted
- 1-level OS- and CS-based analyses dropped to only ~8-10 and 2-3 times the CI contexts
  - From 10-20 and 5-10 times, previously
- Maximum of 33.8 times the CI contexts in the case of 3-level OS analysis (from 13,289, previously)

Equivalent Contexts

- In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted
- 1-level OS- and CS-based analyses dropped to only ~8-10 and 2-3 times the CI contexts
  - From 10-20 and 5-10 times, previously
- Maximum of 33.8 times the CI contexts in the case of 3-level OS analysis (from 13,289, previously)
- ZCWL showed the greatest improvement:

Equivalent Contexts

- In counting the number of equivalent contexts generated in the previous benchmarks, the potential for drastic improvements was highlighted
- 1-level OS- and CS-based analyses dropped to only ~8-10 and 2-3 times the CI contexts
  - From 10-20 and 5-10 times, previously
- Maximum of 33.8 times the CI contexts in the case of 3-level OS analysis (from 13,289, previously)
- ZCWL showed the greatest improvement: from between 2.9×10^4 and 2.1×10^{15} times to only ~3-7

Equivalent Contexts

- Finding equivalent contexts *a priori* would clearly benefit analysis

Equivalent Contexts

- Finding equivalent contexts *a priori* would clearly benefit analysis
- Notes:
  - OS-based analysis generated (~3x) more equivalent contexts, which would likely make it more precise than CSs
  - Longer context strings led to an exponential increase in space required, but only minimal precision improvements
  - ZCWL models cycles insensitively; thus drastically reducing the number of equivalent contexts generated

Distinct Points-to sets

- Can be seen as a rough approximation of the space-efficiency of BDDs

Distinct Points-to sets

- Can be seen as a rough approximation of the space-efficiency of BDDs
- In nearly all types of context abstraction (OS-, CS-, and ZCWL-based), there is no significant advantage over CI analysis

Distinct Points-to sets

- Can be seen as a rough approximation of the space-efficiency of BDDs
- In nearly all types of context abstraction (OS-, CS-, and ZCWL-based), there is no significant advantage over CI analysis
  - However, 1-level context sensitive heap abstractions led to an 11-fold increase.
  - Points-to sets are pairs of abstract objects and contexts, rather than simply the objects themselves

Distinct Points-to sets

- Can be seen as a rough approximation of the space-efficiency of BDDs
- In nearly all types of context abstraction (OS-, CS-, and ZCWL-based), there is no significant advantage over CI analysis
  - However, 1-level context sensitive heap abstractions led to an 11-fold increase.
  - Points-to sets are pairs of abstract objects and contexts, rather than simply the objects themselves
- Representing points-to sets less critical than efficiently representing contexts

Reachable Methods

- Reducing the number of reachable methods allows dead-code elimination

Reachable Methods

- Reducing the number of reachable methods allows dead-code elimination
- Context-sensitive graphs were created, then context “projected” away to enable comparison

Reachable Methods

- Reducing the number of reachable methods allows dead-code elimination
- Context-sensitive graphs were created, then context “projected” away to enable comparison
- Results underwhelming: maximum of 13 methods fewer than CI approach

Reachable Methods

- Reducing the number of reachable methods allows dead-code elimination
- Context-sensitive graphs were created, then context “projected” away to enable comparison
- Results underwhelming: maximum of 13 methods fewer than CI approach
- Results were slightly better for OS-based analysis
  - Node-visitor algorithms where certain types of nodes will never be reached
  - Heap abstractions improve performance on dynamically-allocated objects

Call Edges

- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable

Call Edges

- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable
- As with reachable-method analysis, no significant improvement is detected...

Call Edges

- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable
- As with reachable-method analysis, no significant improvement is detected...
- ...except in `sablecc-j` benchmark w/ 1-level OS

Call Edges

- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable
- As with reachable-method analysis, no significant improvement is detected...
- ...except in `sablecc-j` benchmark w/ 1-level OS
- Benchmark uses tree traversal with numerous `this.getParent()` calls. W/out context, this could generate a huge number of potential call edges

Call Edges

- Measures call graph in terms of number of call edges as opposed to number of reachable methods
  - Again, having fewer call edges is desirable
- As with reachable-method analysis, no significant improvement is detected...
- ...except in `sablecc-j` benchmark w/ 1-level OS
- Benchmark uses tree traversal with numerous `this.getParent()` calls. W/out context, this could generate a huge number of potential call edges
  - 17,925 call edges in CI analysis, only ~5100 in context sensitive test

Virtual Call Resolution

- At compile time, virtual calls introduce potential call edges between pointers and any number of targets

Virtual Call Resolution

- At compile time, virtual calls introduce potential call edges between pointers and any number of targets
- Reducing potential polymorphism of call sites reduces the amount of call edges generated
  - In effect, a subset of the call-edge problem, previously

Virtual Call Resolution

- At compile time, virtual calls introduce potential call edges between pointers and any number of targets
- Reducing potential polymorphism of call sites reduces the amount of call edges generated
  - In effect, a subset of the call-edge problem, previously
- Fully resolving a call site (i.e., removing polymorphism) means it can be replaced by cheaper static methods, allowing further optimization

Virtual Call Resolution

- Again, relatively small improvements

Virtual Call Resolution

- Again, relatively small improvements
- CS-based optimization performs as well as, but never better, than OS-based

Virtual Call Resolution

- Again, relatively small improvements
- CS-based optimization performs as well as, but never better, than OS-based
- Once again, `sablecc-j` provides a good example
  - Some devirtualization can be handled by any context-sensitive analysis
  - A further set of devirtualization requires OS
  - A final set requires context-sensitive heap objects

Cast Safety

- In an OO language, a cast *cannot* fail if the pointer that it is casting can *only* point to variables that are subtypes of the cast

Cast Safety

- In an OO language, a cast *cannot* fail if the pointer that it is casting can *only* point to variables that are subtypes of the cast
  - Presumably, proving that certain casts cannot fail reduces the number of exceptional call edges (Comments?)

Cast Safety

- Once again, we see modest improvements with context sensitivity, particularly with OS analysis and context sensitive heap abstractions

Cast Safety

- Once again, we see modest improvements with context sensitivity, particularly with OS analysis and context sensitive heap abstractions
- In `polyglot` benchmark, the number of potentially failing casts is reduced from 3539 (CI) to 1017.

Cast Safety

- Once again, we see modest improvements with context sensitivity, particularly with OS analysis and context sensitive heap abstractions
- In `polyglot` benchmark, the number of potentially failing casts is reduced from 3539 (CI) to 1017.
- This benchmark involves a large class hierarchy, in which each subclass implements a copy() method
- Using OS, receiver objects performing the casts can be determined, and cast safety made more precise

Cast Safety

- Once again, we see modest improvements with context sensitivity, particularly with OS analysis and context sensitive heap abstractions
- In `polyglot` benchmark, the number of potentially failing casts is reduced from 3539 (CI) to 1017.
- This benchmark involves a large class hierarchy, in which each subclass implements a copy() method
- Using OS, receiver objects performing the casts can be determined, and cast safety made more precise
- Further, OS heap abstractions can more accurately model casts in dynamically-allocated objects

Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes

Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes, *and now we have seen how*

Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes, *and now we have seen how*
- Generally, context-sensitive points-to analysis:
  - improved call-graph precision slightly
  - improved virtual call resolution even more
  - led to major precision improvements in cast safety analysis

Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes, *and now we have seen how*
- Generally, context-sensitive points-to analysis:
  - improved call-graph precision slightly
  - improved virtual call resolution even more
  - led to major precision improvements in cast safety analysis
- OS-based approaches were never less precise than CS-based ones, and scaled better than the latter when context string length was increased

Conclusion

- Is context-sensitive points-to analysis worth it?
  - Yes, *and now we have seen how*
- Generally, context-sensitive points-to analysis:
  - improved call-graph precision slightly
  - improved virtual call resolution even more
  - led to major precision improvements in cast safety analysis
- OS-based approaches were never less precise than CS-based ones, and scaled better than the latter when context string length was increased
- The ZCWL algorithm was never more precise than OS

Conclusion

- Extending the context strings for OS-based analysis gave only modest performance increases, but context-sensitive heaps gave, in the best case, significant improvements

Conclusion

- Extending the context strings for OS-based analysis gave only modest performance increases, but context-sensitive heaps gave, in the best case, significant improvements
- However, efficiently implementing 1H-object-sensitive analysis without BDDs requires further work

Critique

- Discussion of ZCWL algorithm and context-sensitive heaps would benefit the reader
- Are benchmarks particularly suited to OS-based analysis? Are there no benchmarks for which a CS-based approach would show greater improvement?


The End