Compare Less, Defer More

Scaling Value-Contexts Based Whole-Program Heap Analyses

Manas Thakur and V Krishna Nandivada

CC 2019

Heap analysis

- Any analysis that statically approximates information about the runtime heap of a program.
- Usually involves points-to information: which variables may point to which heap locations.
- Examples: (Thread-)escape analysis, shape analysis, interprocedural control-flow analysis.

Context-sensitivity

Analyze a method in each different context from which it is called.

- Call-string based
- Object-sensitive
- Type-sensitive

Context-sensitivity

Analyze a method in each different context from which it is called.

- Call-string based
- Object-sensitive
- Type-sensitive

Compared to context-*insensitive* analyses:

- Usually more precise

Context-sensitivity

Analyze a method in each different context from which it is called.

- Call-string based
- Object-sensitive
- Type-sensitive

Compared to context-*insensitive* analyses:

- Usually more precise
- Usually unscalable

Call-string based context-sensitivity

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){...}
}
```

Call-string based context-sensitivity

```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){...}
15.}
```

- 2 contexts for bar
  - foo_5
  - foo_6

Call-string based context-sensitivity

```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){...}
15.}
```

- 2 contexts for bar
  - foo_5
  - foo_6
- 4 contexts for fb
  - foo_5+bar_11
  - foo_5+bar_12
  - foo_6+bar_11
  - foo_6+bar_12

Call-string based context-sensitivity

```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){...}
15.}
```

- 2 contexts for bar
  - foo_5
  - foo_6
- 4 contexts for fb
  - foo_5+bar_11
  - foo_5+bar_12
  - foo_6+bar_11
  - foo_6+bar_12
- In case of recursion?

Value-contexts[1]

- Contexts defined in terms of data-flow values at call-sites.

[1] Uday P. Khedker and Bageshri Karkare. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. *CC 2008*.

Value-contexts[1]

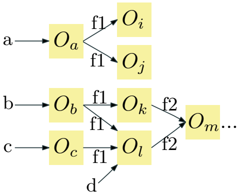
- Contexts defined in terms of data-flow values at call-sites.
- If the lattice of data-flow values is finite, termination is guaranteed.
- Restrict the unbounded length of call-strings without sacrificing precision.

[1] Uday P. Khedker and Bageshri Karkare. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. *CC 2008*.

Value-contexts: Example

```
1.  class A {
2.    A f1,f2;
3.    void foo(){
4.      A a,b,c,d;...
5.      c.bar(a);
6.      d.bar(b);
7.    }
8.    void bar(A p){
9.      A x = new A();
10.     p.f1.f2 = x;
11.     p.fb();
12.     p.fb();
13.   }
14.   void fb(){...}
15.}
```

Points-to graph

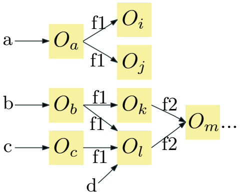


(Line 5)

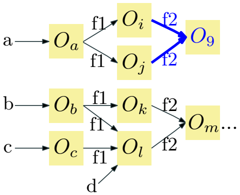
Value-contexts: Example

```
1.  class A {
2.    A f1,f2;
3.    void foo(){
4.      A a,b,c,d;...
5.      c.bar(a);
6.      d.bar(b);
7.    }
8.    void bar(A p){
9.      A x = new A();
10.     p.f1.f2 = x;
11.     p.fb();
12.     p.fb();
13.   }
14.   void fb(){...}
15.}
```

Points-to graph



(Line 5)

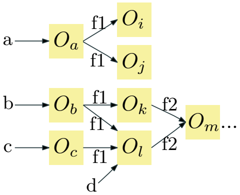


(Line 6)

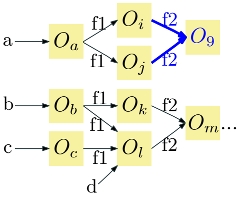
Value-contexts: Example

```
1.  class A {
2.    A f1,f2;
3.    void foo(){
4.      A a,b,c,d;...
5.      c.bar(a);
6.      d.bar(b);
7.    }
8.    void bar(A p){
9.      A x = new A();
10.     p.f1.f2 = x;
11.     p.fb();
12.     p.fb();
13.   }
14.   void fb(){...}
15.}
```

Points-to graph

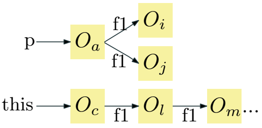


(Line 5)



(Line 6)

Value-context

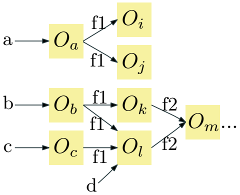


(Line 5)

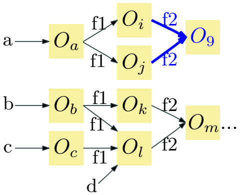
Value-contexts: Example

```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     A a,b,c,d;...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){...}
15.}
```

Points-to graph

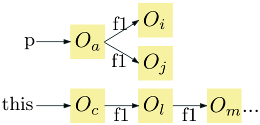


(Line 5)

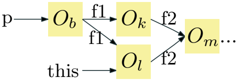


(Line 6)

Value-context



(Line 5)



(Line 6)

Value-contexts in practice: Escape analysis

- We tried using value-contexts to perform whole-program escape analysis for widely used Java benchmarks.

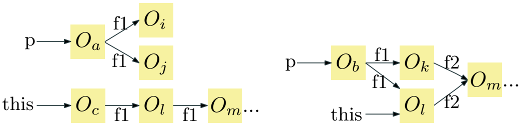
Value-contexts in practice: Escape analysis

- We tried using value-contexts to perform whole-program escape analysis for widely used Java benchmarks.
- For moldyn (the smallest benchmark):
  - Analysis did not terminate in 3 hours!
  - Memory consumed at that time: 373 GB!

Problems with value-contexts

Problem 1: Too much comparison

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){...}
}
```



Graph isomorphism is costly (NP).

Insight 1: Relevance

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- The points-to graph reachable only till `p.f1` is *relevant* for bar (rest is not *accessed*).

Insight 1: Relevance

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- The points-to graph reachable only till `p.f1` is *relevant* for bar (rest is not *accessed*).

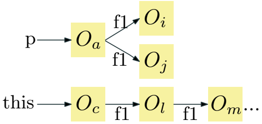
- **Proposal:**
  Identify and use relevant value-contexts.

Insight 1: Example

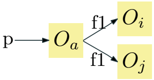
```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){
15.    /*Doesn't access
16.    caller's heap*/
17.  }
18.}
```

- The points-to graph reachable only from `p.f1` is *relevant* for `bar`.
- **Proposal:**
  Identify and use relevant value-contexts.

Line 5:



Value-context



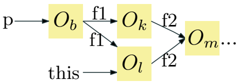
Relevant value-context

Insight 1: Example

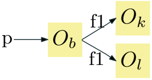
```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){
15.    /*Doesn't access
16.    caller's heap*/
17.  }
18.}
```

- The points-to graph reachable only from `p.f1` is *relevant* for `bar`.
- **Proposal:**
  Identify and use relevant value-contexts.

Line 6:



Value-context



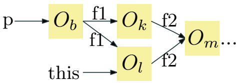
Relevant value-context

Insight 1: Example

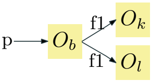
```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){
15.    /*Doesn't access
16.    caller's heap*/
17.  }
18.}
```

- The points-to graph reachable only from `p.f1` is *relevant* for `bar`.
- **Proposal:**
  Identify and use relevant value-contexts.

Line 6:



Value-context



Relevant value-context

Result:
Graphs to be stored/compared significantly smaller.

Problem 2: Too many contexts

- Analyzing a method and maintaining summaries in each context consumes time and memory.

Problem 2: Too many contexts

- Analyzing a method and maintaining summaries in each context consumes time and memory.
- The lattice of points-to graphs is large.

Problem 2: Too many contexts

- Analyzing a method and maintaining summaries in each context consumes time and memory.
- The lattice of points-to graphs is large.
- More contexts also imply comparison with more values at call-sites.

Insight 2a: Level-summarization

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- For a given analysis, even if the relevant value-context changes, the analysis-result may not be affected.

Insight 2a: Level-summarization

```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){
15.    /*Doesn't access
16.    caller's heap*/
17.  }
18.}
```

- For a given analysis, even if the relevant value-context changes, the analysis-result may not be affected.
- For `bar`, O_9 escapes only if the object(s) pointed-to by `p` or `p.f1` escape.

Insight 2a: Level-summarization

```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){
15.    /*Doesn't access
16.    caller's heap*/
17.  }
18.}
```

- For a given analysis, even if the relevant value-context changes, the analysis-result may not be affected.
- For `bar`, O_9 escapes only if the object(s) pointed-to by `p` or `p.f1` escape.
- **Proposal:** Compare only the level-summarized relevant value (LSRV-) contexts.

Insight 2a: Example

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

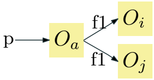
Proposal: Use LSRV-contexts.

Insight 2a: Example

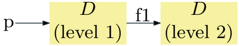
```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){
15.    /*Doesn't access
16.    caller's heap*/
17.  }
18.}
```

Proposal: Use LSRV-contexts.

Line 5:



Relevant value-context



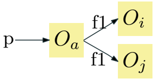
LSRV-context

Insight 2a: Example

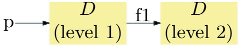
```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){
15.    /*Doesn't access
16.    caller's heap*/
17.  }
18.}
```

Proposal: Use LSRV-contexts.

Line 5:

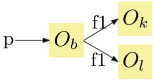


Relevant value-context

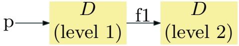


LSRV-context

Line 6:



Relevant value-context



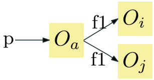
LSRV-context

Insight 2a: Example

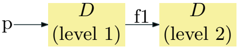
```
1. class A {
2.   A f1,f2;
3.   void foo(){
4.     ...
5.     c.bar(a);
6.     d.bar(b);
7.   }
8.   void bar(A p){
9.     A x = new A();
10.    p.f1.f2 = x;
11.    p.fb();
12.    p.fb();
13.  }
14.  void fb(){
15.    /*Doesn't access
16.    caller's heap*/
17.  }
18.}
```

Proposal: Use LSRV-contexts.

Line 5:

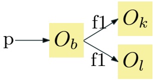


Relevant value-context

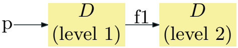


LSRV-context

Line 6:



Relevant value-context



LSRV-context

Result: bar analyzed only once!

Insight 2b: Caller-ignorable

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- Method `fb` is *caller-ignorable*.
  - Caller doesn't need `fb`'s analysis.
  - `fb` can be analyzed separately.

Insight 2b: Caller-ignorable

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

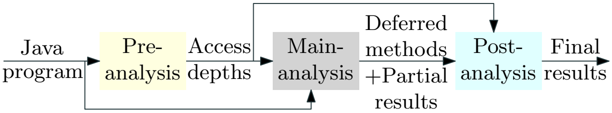
- Method fb is *caller-ignorable*.
  - Caller doesn't need fb's analysis.
  - fb can be analyzed separately.
- **Proposal:**
  Defer the analysis of caller-ignorable methods, and analyze them context-sensitively in a post-pass.

Insight 2b: Caller-ignorable

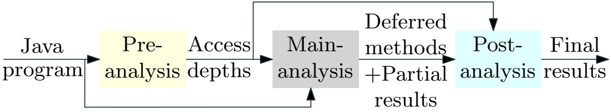
```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- Method `fb` is *caller-ignorable*.
  - Caller doesn't need `fb`'s analysis.
  - `fb` can be analyzed separately.
- **Proposal:**
  Defer the analysis of caller-ignorable methods, and analyze them context-sensitively in a post-pass.
- **Result:**
  Time and memory saved during the costly whole-program analysis.

Proposed approach

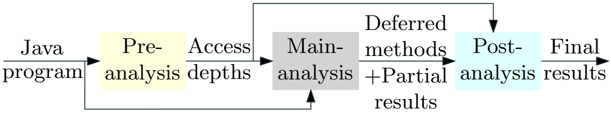


Proposed approach



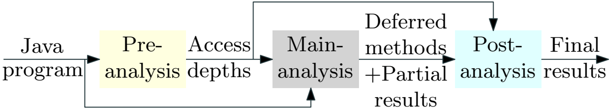
1. Pre-analysis
    - Flow-insensitive, interprocedural – fast.
    - For each method, compute the *access-depth* for each parameter.

Proposed approach



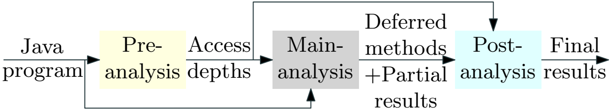
1. Pre-analysis
    - Flow-insensitive, interprocedural – fast.
    - For each method, compute the *access-depth* for each parameter.
2. Main-analysis
    - Context- and flow-sensitive.
    - Compare only LSRV-contexts and defer caller-ignorable methods.

Proposed approach



1. Pre-analysis
   - Flow-insensitive, interprocedural – fast.
   - For each method, compute the *access-depth* for each parameter.
2. Main-analysis
   - Context- and flow-sensitive.
   - Compare only LSRV-contexts and defer caller-ignorable methods.
3. Post-analysis
   - Analyze deferred methods context-sensitively.

Proposed approach



1. Pre-analysis
   - Flow-insensitive, interprocedural – fast.
   - For each method, compute the *access-depth* for each parameter.
2. Main-analysis
   - Context- and flow-sensitive.
   - Compare only LSRV-contexts and defer caller-ignorable methods.
3. Post-analysis
   - Analyze deferred methods context-sensitively.

Detailed algorithms in the paper.

Instantiations

1. Escape analysis
   - Dataflow values: $\{$DoesNotEscape (D), Escapes (E)$\}$.
   - Meet: $D \sqcap D = D$, $D \sqcap E = E \sqcap D = E \sqcap D = E$.

2. Control-flow analysis
   - Find the types that can flow into each variable.
   - Applications: call-graph construction, typecast checks, etc.
   - Dataflow values: Set of all classes in the program.
   - Meet: Union.

Evaluation

Experimental setup

- Implementation: Soot optimization framework
- Runtime: OpenJDK HotSpot JVM v8
- System: 2.3 GHz AMD with 64 cores and 512 GB RAM
- Benchmarks: DaCapo 9.12 and JGF

Versions compared

- B: Base
  - Escape analysis[2]
  - Control-flow analysis[3]

[2](Value-contexts implementation of) John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. *OOPSLA 1999*.

[3]Rohan Padhye and Uday P. Khedker. Interprocedural Data Flow Analysis in Soot Using Value Contexts. *SOAP 2013*.

Versions compared

- B: Base
  - Escape analysis[2]
  - Control-flow analysis[3]
- OM: Only Main (i.e., no trimming of value-contexts)

[2](Value-contexts implementation of) John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. *OOPSLA 1999*.

[3]Rohan Padhye and Uday P. Khedker. Interprocedural Data Flow Analysis in Soot Using Value Contexts. *SOAP 2013*.

Versions compared

- B: Base
  - Escape analysis[2]
  - Control-flow analysis[3]
- OM: Only Main (i.e., no trimming of value-contexts)
- PM: Pre and Main (i.e., no deferring of methods)

[2](Value-contexts implementation of) John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. *OOPSLA 1999*.

[3]Rohan Padhye and Uday P. Khedker. Interprocedural Data Flow Analysis in Soot Using Value Contexts. *SOAP 2013*.

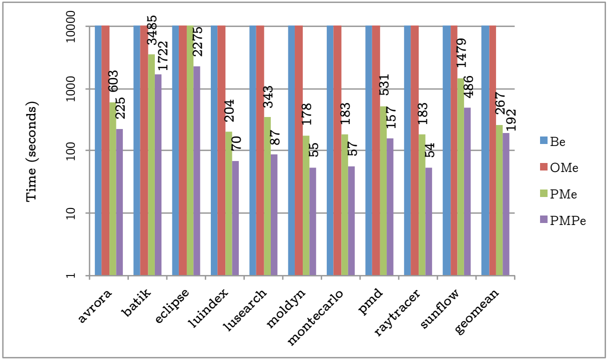
Versions compared

- B: Base
  - Escape analysis[2]
  - Control-flow analysis[3]
- OM: Only Main (i.e., no trimming of value-contexts)
- PM: Pre and Main (i.e., no deferring of methods)
- PMP: Pre, Main and Post (i.e., the full proposed version)

[2](Value-contexts implementation of) John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. *OOPSLA 1999*.

[3]Rohan Padhye and Uday P. Khedker. Interprocedural Data Flow Analysis in Soot Using Value Contexts. *SOAP 2013*.

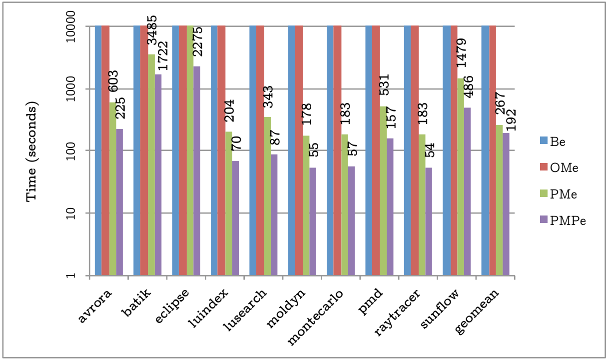
Analysis time: Escape analysis



- B_e: Base
- OM_e: Only Main
- PM_e: Pre and Main
- PMP_e: Pre, Main and Post

- B_e and OM_e do not terminate for any benchmark.

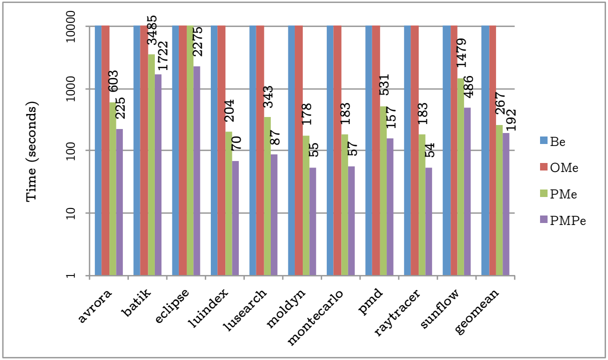
Analysis time: Escape analysis



- B_e: Base
- OM_e: Only Main
- PM_e: Pre and Main
- PMP_e: Pre, Main and Post

- B_e and OM_e do not terminate for any benchmark.
- PM_e scales better, but still does not terminate for eclipse.

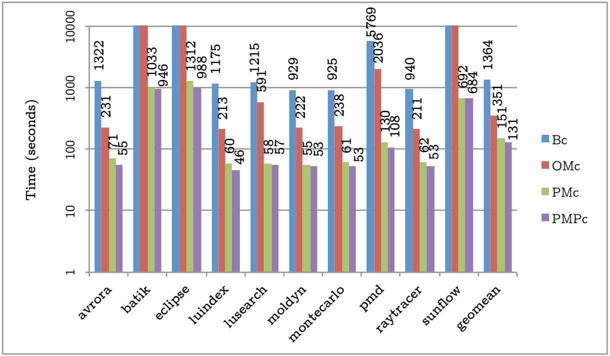
Analysis time: Escape analysis



- B_e: Base
- OM_e: Only Main
- PM_e: Pre and Main
- PMP_e: Pre, Main and Post

- B_e and OM_e do not terminate for any benchmark.
- PM_e scales better, but still does not terminate for eclipse.
- With just ∼2 seconds for the pre and post analyses, PMP_e scales for all benchmarks (average ∼28% over PM_e).

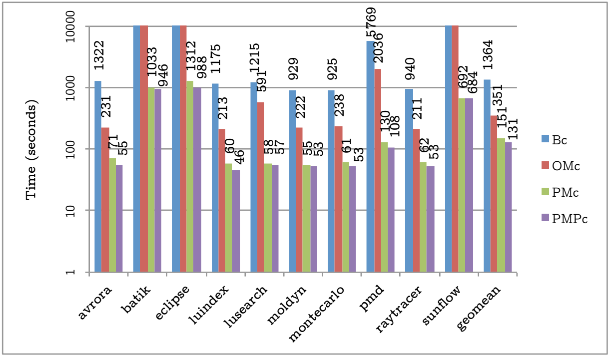
Analysis time: Control-flow analysis



- B_c: Base
- OM_c: Only Main
- PM_c: Pre and Main
- PMP_c: Pre, Main and Post

- B_c and OM_c do not terminate for three large benchmarks.

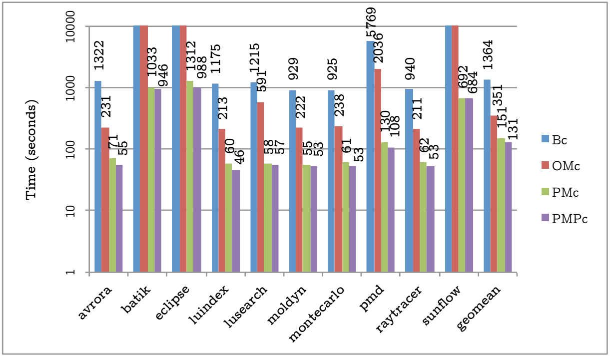
Analysis time: Control-flow analysis



- B_c: Base
- OM_c: Only Main
- PM_c: Pre and Main
- PMP_c: Pre, Main and Post

- B_c and OM_c do not terminate for three large benchmarks.
- PM_c scales over all benchmarks.

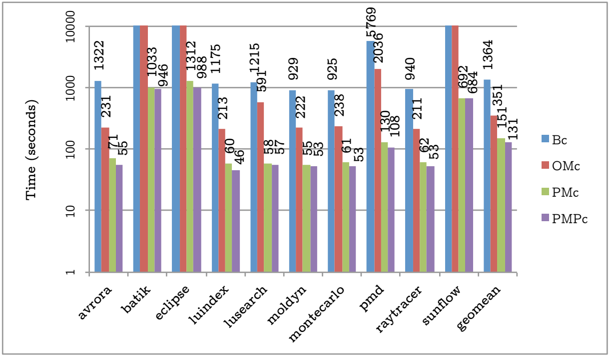
Analysis time: Control-flow analysis



- B_c: Base
- OM_c: Only Main
- PM_c: Pre and Main
- PMP_c: Pre, Main and Post

- B_c and OM_c do not terminate for three large benchmarks.
- PM_c scales over all benchmarks.
- PMP_c improves over B_c by $\sim$90%, and over PM_c by $\sim$14% (average).

Analysis time: Control-flow analysis



- B_c: Base
- OM_c: Only Main
- PM_c: Pre and Main
- PMP_c: Pre, Main and Post

- B_c and OM_c do not terminate for three large benchmarks.
- PM_c scales over all benchmarks.
- PMP_c improves over B_c by $\sim$90%, and over PM_c by $\sim$14% (average).

Otherwise unanalyzable benchmarks in less than 40 minutes.

Peak memory consumption

Benchmark	Memory (GB)			
	B_e	PMP_e	B_c	PMP_c
avrora	-	21	54	11
batik	-	45	-	64
eclipse	-	57	-	49
luindex	-	6	58	11
lusearch	-	10	54	11
pmd	-	11	127	13
sunflow	-	21	-	53
moldyn	-	6	29	11
montecarlo	-	6	29	9
raytracer	-	6	29	10
geomean	-	13	47	18

Peak memory consumption

Benchmark	Memory (GB)			
	B_e	PMP_e	B_c	PMP_c
avrora	-	21	54	11
batik	-	45	-	64
eclipse	-	57	-	49
luindex	-	6	58	11
lusearch	-	10	54	11
pmd	-	11	127	13
sunflow	-	21	-	53
moldyn	-	6	29	11
montecarlo	-	6	29	9
raytracer	-	6	29	10
geomean	-	13	47	18

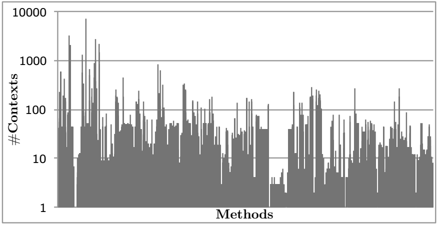
- Earlier, systems with very large memories (~512GB) were not enough.
- Now, a 32-64 GB machine should be sufficient.

Number of contexts

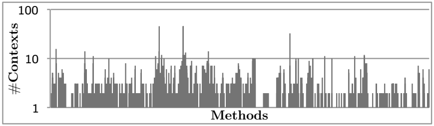
Benchmark	Average #contexts			
	B_e	PMP_e	B_c	PMP_c
avrora	-	1.4	9.5	1.2
batik	-	1.4	-	1.3
eclipse	-	1.9	-	1.4
luindex	-	1.3	10.6	1.2
lusearch	-	1.3	10.5	1.2
pmd	-	1.3	11.9	1.2
sunflow	-	1.3	-	1.2
moldyn	-	1.3	9.5	1.3
montecarlo	-	1.3	9.4	1.2
raytracer	-	1.3	9.4	1.2
geomean	-	1.4	10.1	1.2

Number of contexts

Bench-mark	Average #contexts			
	B_e	PMP_e	B_c	PMP_c
avrora	-	1.4	9.5	1.2
batik	-	1.4	-	1.3
eclipse	-	1.9	-	1.4
luindex	-	1.3	10.6	1.2
lusearch	-	1.3	10.5	1.2
pmd	-	1.3	11.9	1.2
sunflow	-	1.3	-	1.2
moldyn	-	1.3	9.5	1.3
montecarlo	-	1.3	9.4	1.2
raytracer	-	1.3	9.4	1.2
geomean	-	1.4	10.1	1.2



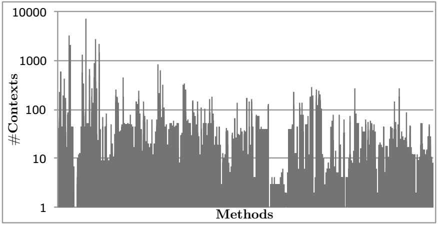
pmd-B_c



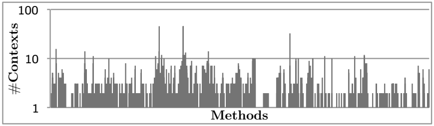
pmd-PMP_c

Number of contexts

Benchmark	Average #contexts			
	B_e	PMP_e	B_c	PMP_c
avrora	-	1.4	9.5	1.2
batik	-	1.4	-	1.3
eclipse	-	1.9	-	1.4
luindex	-	1.3	10.6	1.2
lusearch	-	1.3	10.5	1.2
pmd	-	1.3	11.9	1.2
sunflow	-	1.3	-	1.2
moldyn	-	1.3	9.5	1.3
montecarlo	-	1.3	9.4	1.2
raytracer	-	1.3	9.4	1.2
geomean	-	1.4	10.1	1.2



pmd-B_c

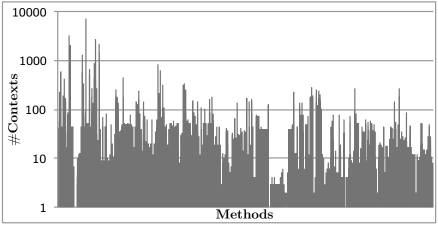


pmd-PMP_c

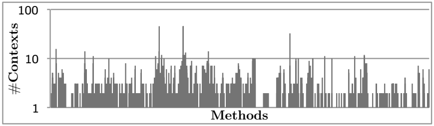
Significant reduction in #contexts

Number of contexts

Benchmark	Average #contexts			
	B_e	PMP_e	B_c	PMP_c
avrora	-	1.4	9.5	1.2
batik	-	1.4	-	1.3
eclipse	-	1.9	-	1.4
luindex	-	1.3	10.6	1.2
lusearch	-	1.3	10.5	1.2
pmd	-	1.3	11.9	1.2
sunflow	-	1.3	-	1.2
moldyn	-	1.3	9.5	1.3
montecarlo	-	1.3	9.4	1.2
raytracer	-	1.3	9.4	1.2
geomean	-	1.4	10.1	1.2



pmd-B_c

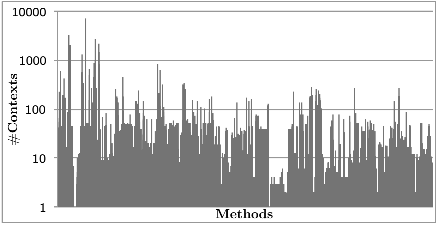


pmd-PMP_c

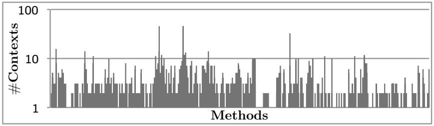
Significant reduction in #contexts ⇒ Significant reduction in resources spent

Number of contexts

Bench-mark	Average #contexts			
	B_e	PMP_e	B_c	PMP_c
avrora	-	1.4	9.5	1.2
batik	-	1.4	-	1.3
eclipse	-	1.9	-	1.4
luindex	-	1.3	10.6	1.2
lusearch	-	1.3	10.5	1.2
pmd	-	1.3	11.9	1.2
sunflow	-	1.3	-	1.2
moldyn	-	1.3	9.5	1.3
montecarlo	-	1.3	9.4	1.2
raytracer	-	1.3	9.4	1.2
geomean	-	1.4	10.1	1.2



pmd-B_c

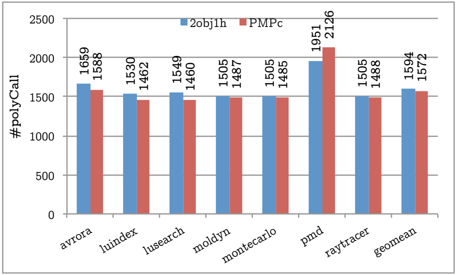


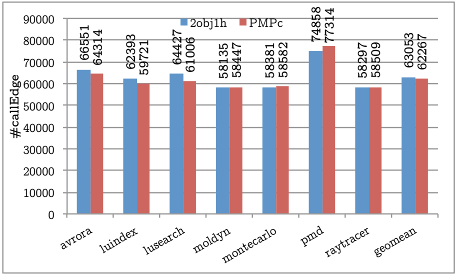
pmd-PMP_c

Significant reduction in #contexts ⇒ Significant reduction in resources spent ⇒ **Scalability.**

Comparison with 2obj1h

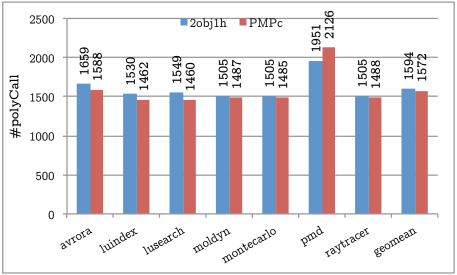
Comparison with 2obj1h (lower the better)

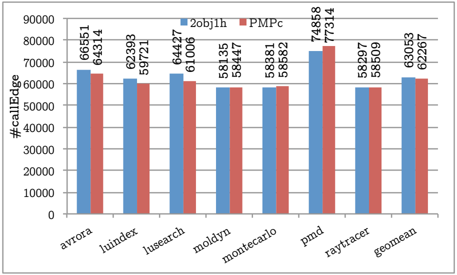




- Precision: comparable.

Comparison with 2obj1h (lower the better)





- Precision: comparable.
- Scalability:
  - 2obj1h did not terminate for batik, eclipse and sunflow.
  - For the rest: LSRV-contexts (PMP_c) took 89.2% lesser time and 59.4% lesser memory.

Conclusion and Future work

Conclusion:

- LSRV-contexts scale whole-program context-sensitive analyses without losing precision.
- Identifying relevance of value-contexts is a novel and effective idea.
- Evaluation on two non-trivial analyses demonstrates the generality.

Conclusion and Future work

Conclusion:

- LSRV-contexts scale whole-program context-sensitive analyses without losing precision.
- Identifying relevance of value-contexts is a novel and effective idea.
- Evaluation on two non-trivial analyses demonstrates the generality.

Future work:

- Study the cases where the precisions of object-sensitive and call-string based approaches differ.
- Add heap-cloning to value-contexts using the scalable approaches proposed as part of LSRV-contexts.

Conclusion and Future work

Conclusion:

- LSRV-contexts scale whole-program context-sensitive analyses without losing precision.
- Identifying relevance of value-contexts is a novel and effective idea.
- Evaluation on two non-trivial analyses demonstrates the generality.

Future work:

- Study the cases where the precisions of object-sensitive and call-string based approaches differ.
- Add heap-cloning to value-contexts using the scalable approaches proposed as part of LSRV-contexts.

Thank you.

Example: Access-depths

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- For bar: $\{\langle \mathtt{this}, 0\rangle, \langle \mathtt{p}, 2\rangle\}$

Example: Access-depths

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- For bar: $\{\langle \mathtt{this}, 0\rangle, \langle \mathtt{p}, 2\rangle\}$

  $\Rightarrow$ Relevant points-to (sub)graph: *ptsto*($\mathtt{p}$), *ptsto*($\mathtt{p.f1}$)

Example: Access-depths

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- For bar: $\{\langle \mathtt{this}, 0\rangle, \langle \mathtt{p}, 2\rangle\}$

  $\Rightarrow$ Relevant points-to (sub)graph: *ptsto*($\mathtt{p}$), *ptsto*($\mathtt{p.f1}$)

- For fb: $\{\langle \mathtt{this}, 0\rangle\}$

Example: Access-depths

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- For `bar`: $\{\langle \mathtt{this}, 0\rangle, \langle \mathtt{p}, 2\rangle\}$

  $\Rightarrow$ Relevant points-to (sub)graph: *ptsto*($\mathtt{p}$), *ptsto*($\mathtt{p.f1}$)

- For `fb`: $\{\langle \mathtt{this}, 0\rangle\}$

  $\Rightarrow$ `fb` is caller-ignorable

Example: Access-depths

```
class A {
  A f1,f2;
  void foo(){
    ...
    c.bar(a);
    d.bar(b);
  }
  void bar(A p){
    A x = new A();
    p.f1.f2 = x;
    p.fb();
    p.fb();
  }
  void fb(){
    /*Doesn't access
    caller's heap*/
  }
}
```

- For bar: $\{\langle \mathtt{this}, 0\rangle, \langle \mathtt{p}, 2\rangle\}$

  $\Rightarrow$ Relevant points-to (sub)graph: *ptsto*(p), *ptsto*(p.f1)

- For fb: $\{\langle \mathtt{this}, 0\rangle\}$

  $\Rightarrow$ fb is caller-ignorable

- Detailed algorithms for pre, main, and post analyses in the paper.

Static characteristics of benchmarks

Benchmark	Application #classes	Application size (MB)	#Referred JDK classes
avrora	527	2.7	1588
batik	1038	6.0	3700
eclipse	1608	14.0	2589
luindex	199	1.3	1485
lusearch	198	1.3	1481
pmd	697	4.1	1607
sunflow	225	1.7	3509
moldyn	13	0.15	1555
montecarlo	19	0.67	1555
raytracer	19	0.21	1555

Static characteristics of benchmarks

Benchmark	Application #classes	Application size (MB)	#Referred JDK classes
avrora	527	2.7	1588
batik	1038	6.0	3700
eclipse	1608	14.0	2589
luindex	199	1.3	1485
lusearch	198	1.3	1481
pmd	697	4.1	1607
sunflow	225	1.7	3509
moldyn	13	0.15	1555
montecarlo	19	0.67	1555
raytracer	19	0.21	1555

Sizes range from 150 KB (small programs) to 14 MB (large applications).

Analysis time: Pre and Post

Benchmark	Analysis time (seconds)		
	Pre	Post_e	Post_c
avrora	1.0	0.4	0.5
batik	2.2	1.8	2.4
eclipse	2.7	6.0	6.1
luindex	1.1	0.4	0.7
lusearch	1.0	0.5	0.9
pmd	1.3	0.4	0.7
sunflow	2.1	1.6	2.2
moldyn	0.9	0.4	0.6
montecarlo	0.9	0.4	0.3
raytracer	0.9	0.4	0.3
geomean	1.3	0.7	0.9

- Pre-analysis common for both the instantiations.

Analysis time: Pre and Post

Benchmark	Analysis time (seconds)		
	Pre	Post_e	Post_c
avrora	1.0	0.4	0.5
batik	2.2	1.8	2.4
eclipse	2.7	6.0	6.1
luindex	1.1	0.4	0.7
lusearch	1.0	0.5	0.9
pmd	1.3	0.4	0.7
sunflow	2.1	1.6	2.2
moldyn	0.9	0.4	0.6
montecarlo	0.9	0.4	0.3
raytracer	0.9	0.4	0.3
geomean	1.3	0.7	0.9

- Pre-analysis common for both the instantiations.
- The time required for both the pre and the post analyses is negligible ($\sim$2 seconds).