Staged Static+Dynamic Partial Analysis for Java-like Languages

SERI 2023 Goa

Aditya Anand

Indian Institute of Technology Mandi

June 2nd, 2023

Content

- Aditya Anand and Manas Thakur. 2022. Principles of Staged Static+Dynamic Partial Analysis. In Static Analysis: 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022.

- Aditya Anand. 2022. A Study of the Impact of Callbacks in Staged Static+Dynamic Partial Analysis. In Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2022).

Introduction

- Several popular languages (such as Java, C#, Scala) use just-in-time (JIT) compilation.

Introduction

- Several popular languages (such as Java, C#, Scala) use just-in-time (JIT) compilation.
- Performing precise program-analysis during JIT Compilation – highly inefficient.

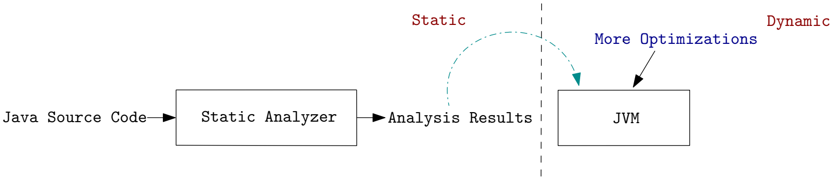
Introduction

- Several popular languages (such as Java, C#, Scala) use just-in-time (JIT) compilation.
- Performing precise program-analysis during JIT Compilation – highly inefficient.
- Most JIT compilers sacrifice precision for efficiency.

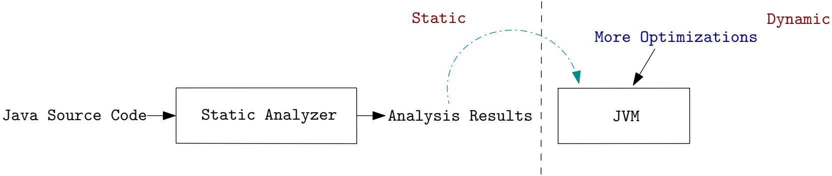
Introduction

- Several popular languages (such as Java, C#, Scala) use just-in-time (JIT) compilation.
- Performing precise program-analysis during JIT Compilation – highly inefficient.
- Most JIT compilers sacrifice precision for efficiency.
- Can we use static analysis to impart precision in JIT analyses?

Using Static Analysis Results in JIT Compilers

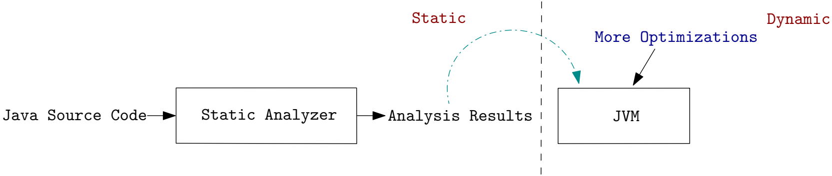


Using Static Analysis Results in JIT Compilers



- Challenge:
  - Library code needed to perform whole-program analysis.

Using Static Analysis Results in JIT Compilers



- Challenge:
  - Library code needed to perform whole-program analysis.
  - Imprecise results due to conservative assumptions.

Partial Program Analysis

Partial Program Analysis

❖ Proposed: OOPSLA 2008

Enabling Static Analysis for Partial Java Programs

Barthélémy Dagenais   Laurie Hendren
McGill University, Montréal, Québec, Canada
[bart,hendren]@cs.mcgill.ca

Partial Program Analysis

- Proposed: OOPSLA 2008

Enabling Static Analysis for Partial Java Programs

Barthélémy Dagenais   Laurie Hendren
McGill University, Montréal, Québec, Canada
[bart,hendren]@cs.mcgill.ca

- Applied to JIT compilers: TOPLAS 2019

PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs

MANAS THAKUR and V. KRISHNA NANDIVADA, IIT Madras

PYE Framework and Conditional Values

"Precise-Yet-Efficient" framework generates highly precise analysis results for JIT compilers at a very low cost:

- Offloads expensive analysis to static compiler (javac) and generates *conditional values*.
- JIT component evaluates the conditional values at run-time and generates final analysis result.

PYE Framework and Conditional Values

"Precise-Yet-Efficient" framework generates highly precise analysis results for JIT compilers at a very low cost:

- Offloads expensive analysis to static compiler (javac) and generates *conditional values*.
- JIT component evaluates the conditional values at run-time and generates final analysis result.

- Essentially, conditional values for a program element enlist the dependencies of that element.

Dependencies in form of Conditional Values

```java
class A {
   void foo(B b) {
      A a1 = new A(); // Object O3
      A a2 = new A(); // Object O4
      a1.bar(a2);
      L l1 = new L(); // Object O6
      l1.lib(a2); }
   void bar(A p1) {
      // no assignment to p1
   } }
```

```java
class L {
   // A library class
   void lib(A r1) {
      // A library method
      ...
   } }
```

Dependencies in form of Conditional Values

```
class A {
   void foo(B b) {
      A a1 = new A(); // Object O3
      A a2 = new A(); // Object O4
      a1.bar(a2);
      L l1 = new L(); // Object O6
      l1.lib(a2); }
   void bar(A p1) {
      // no assignment to p1
   } }
```

```
class L {
   // A library class
   void lib(A r1) {
      // A library
           method
      ...
   } }
```

- Conditional values for object O_4 (for escape analysis):

Dependencies in form of Conditional Values

```
class A {
   void foo(B b) {
      A a1 = new A(); // Object O3
      A a2 = new A(); // Object O4
      a1.bar(a2);
      L l1 = new L(); // Object O6
      l1.lib(a2); }
   void bar(A p1) {
      // no assignment to p1
   } }
```

```
class L {
   // A library class
   void lib(A r1) {
      // A library method
      ...
   } }
```

- Conditional values for object O_4 (for escape analysis):

$$g_{\texttt{A.foo}}(O_4) = \{\langle\langle \texttt{A.bar}, \texttt{p1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{A.bar}, \texttt{p1}\rangle, E, E\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\} \quad (1)$$

Partial Result

$$[\![g_{\texttt{A.foo}}(O_4)]\!] = \ \sqcap_{ea} \{ D, \langle \langle \texttt{L.lib}, \texttt{r1} \rangle, D, D \rangle, D, \langle \langle \texttt{L.lib}, \texttt{r1} \rangle, E, E \rangle \}$$

Partial Result

$$[\![g_{\texttt{A.foo}}(O_4)]\!] = \ \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because f_n(y) \neq v, so = \perp \ i.e\ D)$$

Partial Result

$$[\![g_{\texttt{A.foo}}(O_4)]\!] = \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because f_n(y) \neq v, so =\perp \; i.e \; D)$$

$$[\![g_{\texttt{A.foo}}(O_4)]\!] = \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

Partial Result

$$[\![g_{\texttt{A.foo}}(O_4)]\!] = \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because f_n(y) \neq v, so = \perp\ i.e\ D)$$

$$[\![g_{\texttt{A.foo}}(O_4)]\!] = \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because D \sqcap D = D) \tag{2}$$

Partial Result

$$\llbracket g_{\texttt{A.foo}}(O_4) \rrbracket = \ \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because f_n(y) \neq v, so =\perp \ i.e \ D)$$

$$\llbracket g_{\texttt{A.foo}}(O_4) \rrbracket = \ \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because D \sqcap D = D) \tag{2}$$

❖ Partial Result:

$$f_{\texttt{a.foo}}(O_4) = \ \sqcap_{ea} \ \{\langle\langle \texttt{l.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{l.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

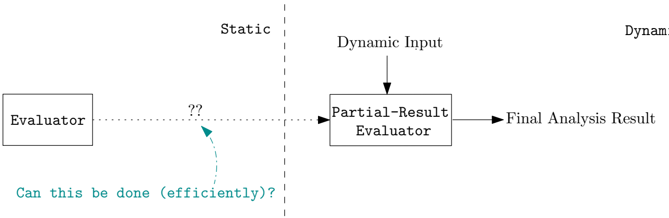
Existing PYE-like Staged Static+JIT Analyses

Existing PYE-like Staged Static+JIT Analyses

- Is it correct to stage whole-program analysis across static and JIT compilation?

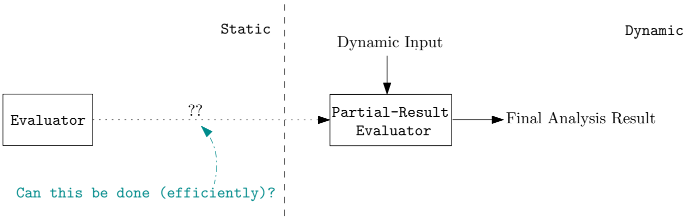
Existing PYE-like Staged Static+JIT Analyses

- Is it correct to stage whole-program analysis across static and JIT compilation?
  - What is the form of the evaluator needed during JIT compilation?



Existing PYE-like Staged Static+JIT Analyses

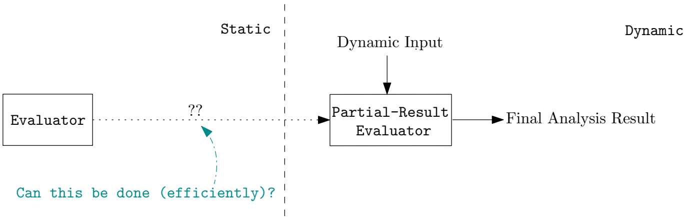
- Is it correct to stage whole-program analysis across static and JIT compilation?
  - What is the form of the evaluator needed during JIT compilation?



- Does the precision of staged analysis remain same as whole-program analysis?

Existing PYE-like Staged Static+JIT Analyses

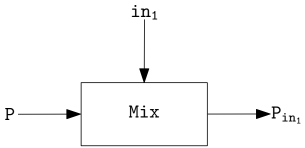
- Is it correct to stage whole-program analysis across static and JIT compilation?
  - What is the form of the evaluator needed during JIT compilation?



- Does the precision of staged analysis remain same as whole-program analysis? What about its efficiency?

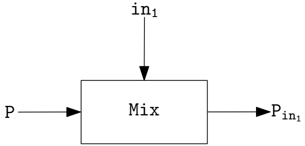
Partial Evaluation

Partial evaluation [Jones 1996] specializes a given program with its statically available inputs. The resultant partially evaluated program can later be executed with the dynamic inputs to generate the final output.



Partial Evaluation

Partial evaluation [Jones 1996] specializes a given program with its statically available inputs. The resultant partially evaluated program can later be executed with the dynamic inputs to generate the final output.



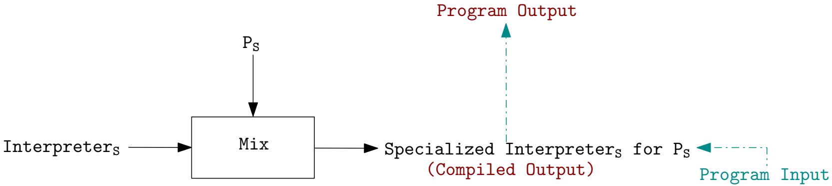
❖ Advantage: The specialized program P_{in_1} often executes faster compared to executing the original program P provided both static and dynamic inputs together.

Futamura Projections

- Partial evaluation has also been used to specialize interpreters and their generators, based on the notion of Futamura projections.

Futamura Projections

❖ Partial evaluation has also been used to specialize interpreters and their generators, based on the notion of Futamura projections.



Modeling Partial Analysis based on Partial Evaluation

Mapped Notation

Source Program	$g_m(x)$ (Conditional Values)
Static Inputs	S_x (Resolved Values)
Runtime Inputs	D_x (Unresolved Values)
Evaluator	$\mathtt{CEval}_\psi$

Conditional Value and Evaluator

$\langle$A foo LOCAL 4 $\langle$nil$\rangle\rangle$:

$\{\langle\langle$STATIC A bar PARM 1$\langle$nil$\rangle\rangle$ D D$\rangle$, $\langle\langle$DYNAMIC L lib RETVAL 1$\langle$nil$\rangle$ D D$\rangle$,
$\langle\langle$STATIC A bar PARM 1$\langle$nil$\rangle\rangle$ E E$\rangle$, $\langle\langle$DYNAMIC L lib RETVAL 1$\langle$nil$\rangle$ E E$\rangle\}$

Conditional Value and Evaluator

$\langle$A foo LOCAL 4 $\langle$nil$\rangle\rangle$:

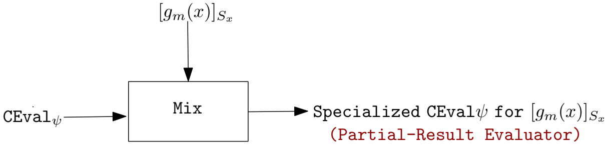
$\{\langle\langle$STATIC A bar PARM 1$\langle$nil$\rangle\rangle$ D D$\rangle$, $\langle\langle$DYNAMIC L lib RETVAL 1$\langle$nil$\rangle$ D D$\rangle$,
$\langle\langle$STATIC A bar PARM 1$\langle$nil$\rangle\rangle$ E E$\rangle$, $\langle\langle$DYNAMIC L lib RETVAL 1$\langle$nil$\rangle$ E E$\rangle\}$

```
Procedure CEval(g_m(x), IN_{g_m(x)})
    Initialize a list L of statically known dependencies.
    foreach d ∈ IN_{gm(x)} do
        Add d to L.
        Add the transitive dependencies of d to L.
    Form strongly connected components (SCCs) in the list L.
    repeat
        foreach strongly connected component S formed above do
            if ∄e ∈ S s.t. e depends on another SCC then
                ∀e ∈ S, resolve e to ⊥.
            Take a meet of the resolved values in each SCC
    until fixed point;
```

AM Projections

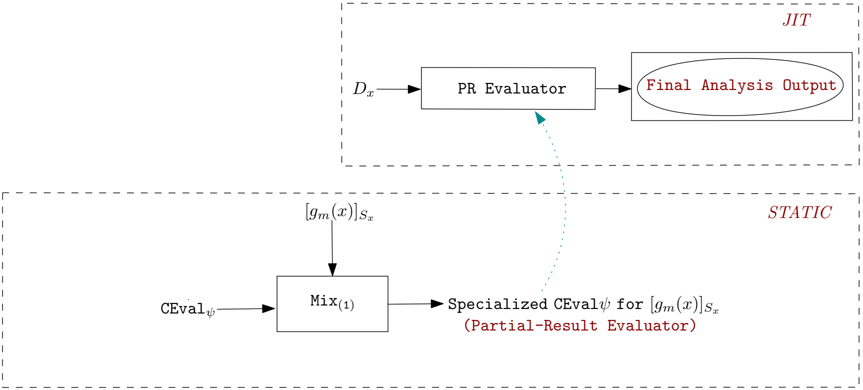
AM Projections

❖ Specialize the evaluator with the partial result.



AM Projections

❖ Specialize the evaluator with the partial result.



Generated Partial Result Evaluator

```
class PartialResultEvaluator {
   public static void main(String[] args) {
      // Read the values for dynamic dependencies
      x1 = Resolved value of <L.lib, r1> // first dependence
      x2 = Resolved value of <L.lib, r1> // second dependence
      res = x1 ⊓_ea x2
      print(res);
   }
}
```

Schema of the partial-result evaluator emitted for $g_{\texttt{A.foo}}(O_4)$.

Generated Partial Result Evaluator

```
class PartialResultEvaluator {
   public static void main(String[] args) {
      // Read the values for dynamic dependencies
      x1 = Resolved value of <L.lib, r1> // first dependence
      x2 = Resolved value of <L.lib, r1> // second dependence
      res = x1 ⊓_ea x2
      print(res);
   }
}
```

Schema of the partial-result evaluator emitted for $g_{\texttt{A.foo}}(O_4)$.

- Can be placed in any VM to obtain the final analysis result for O_4.

Efficiency, Correctness and Precision of Staging

Efficiency, Correctness and Precision of Staging

❖ Lemma 1. Statically available Input

If the set of statically available dependencies is empty, then the specialization performed by the first AM projection for a conditional-value evaluator can be seen in same light as the specialization performed by the first Futamura projection for a program interpreter.

Efficiency, Correctness and Precision of Staging

❖ Lemma 1. Statically available Input

If the set of statically available dependencies is empty, then the specialization performed by the first AM projection for a conditional-value evaluator can be seen in same light as the specialization performed by the first Futamura projection for a program interpreter.

❖ Lemma 2. Maximal Specialization

Partial evaluation of a program with a statically available input implies that the program is specialized to the extent possible (that is, maximally specialized) with respect to that input.

Efficiency, Correctness and Precision of Staging

❖ Theorem 1. Efficiency

For a given program element and its statically available dependencies, the partial-result evaluator obtained by the first AM projection is maximal in terms of the conditional-value evaluation that can be performed statically.

Efficiency, Correctness and Precision of Staging

- Theorem 1. Efficiency

For a given program element and its statically available dependencies, the partial-result evaluator obtained by the first AM projection is maximal in terms of the conditional-value evaluation that can be performed statically.

- Theorem 2. Precision and Correctness

For any program element, the analysis results generated by a whole-program analysis and by the corresponding staged analysis are the same.

Challenges with Callbacks

Challenges with Callbacks

- Callbacks: Library method calls an application method.

Challenges with Callbacks

❖ Callbacks: Library method calls an application method.

```
class A {
  void foo(X p) {
    Y m = new Y();
         // Object O3
    p.f = m;
    this.bar(m);
  }
  void bar(Y q) {
    Z r = new Z();
         // Object O8
    q.g = r;
  }
} /* class A */
```

Challenges with Callbacks

❖ Callbacks: Library method calls an application method.

```
class A {
  void foo(X p) {
    Y m = new Y();
          // Object O3
    p.f = m;
    this.bar(m);
  }
  void bar(Y q) {
    Z r = new Z();
          // Object O8
    q.g = r;
  }
} /* class A */
```

Conditional Value for Object:

O_3: {⟨caller,⟨argument,1⟩, ⟨A.bar,⟨parameter,1⟩⟩}

Challenges with Callbacks

- Callbacks: Library method calls an application method.

```
class A {
  void foo(X p) {
    Y m = new Y();
          // Object O3
    p.f = m;
    this.bar(m);
  }
  void bar(Y q) {
    Z r = new Z();
          // Object O8
    q.g = r;
  }
} /* class A */
```

Conditional Value for Object:

O_3: $\{\langle \texttt{caller}, \langle \texttt{argument}, 1\rangle, \langle \text{A.bar}, \langle \texttt{parameter}, 1\rangle\rangle\}$

O_8: $\{\langle \texttt{caller}, \langle \texttt{argument}, 1\rangle\rangle\}$

Challenges with Callbacks

- Callbacks: Library method calls an application method.

```
class A {
  void foo(X p) {
    Y m = new Y();
           // Object O3
    p.f = m;
    this.bar(m);
  }
  void bar(Y q) {
    Z r = new Z();
           // Object O8
    q.g = r;
  }
} /* class A */
```

Conditional Value for Object:

O_3: $\{\langle caller, \langle argument, 1\rangle, \langle A.bar, \langle parameter, 1\rangle\rangle\}$

O_8: $\{\langle caller, \langle argument, 1\rangle\rangle\}$

foo: Library overridden Method?

Challenges with Callbacks

- Callbacks: Library method calls an application method.

```
class A {
  void foo(X p) {
    Y m = new Y();
          // Object O3
    p.f = m;
    this.bar(m);
  }
  void bar(Y q) {
    Z r = new Z();
          // Object O8
    q.g = r;
  }
} /* class A */
```

Conditional Value for Object:

O_3: $\{\langle caller, \langle argument, 1\rangle, \langle A.bar, \langle parameter, 1\rangle\rangle\}$

O_8: $\{\langle caller, \langle argument, 1\rangle\rangle\}$

foo: Library overridden Method?

- Will generate unsound results.

Methods and Objects affected by Callbacks

Bench-mark	Callback methods	Total methods	Callback objects	Total objects
avrora	43	3181	147	13344
batik	896	6934	1865	34137
lusearch	123	1971	338	9054
luindex	69	1998	171	10260
pmd	595	5941	1301	30016
sunflow	49	2428	104	14136
h2	639	4777	1315	27610
xalan	821	6396	2131	31488
fop	968	11470	2387	74590
eclipse	1515	29419	3505	79443

Methods and Objects affected by Callbacks

Benchmark	Callback methods	Total methods	Callback objects	Total objects
avrora	43	3181	147	13344
batik	896	6934	1865	34137
lusearch	123	1971	338	9054
luindex	69	1998	171	10260
pmd	595	5941	1301	30016
sunflow	49	2428	104	14136
h2	639	4777	1315	27610
xalan	821	6396	2131	31488
fop	968	11470	2387	74590
eclipse	1515	29419	3505	79443

- On an average 7.7% of methods and 4.1% of objects are affected by callbacks.

Conclusion

- Formalized the theory of staged static+dynamic partial analysis based on the theory of partial evaluation.

Conclusion

- Formalized the theory of staged static+dynamic partial analysis based on the theory of partial evaluation.
- Projections for efficiently generating the evaluators (and their generators) for partial results.

Conclusion

- Formalized the theory of staged static+dynamic partial analysis based on the theory of partial evaluation.
- Projections for efficiently generating the evaluators (and their generators) for partial results.
- Proved the correctness and precision of staged static+dynamic analysis.

Conclusion

- Formalized the theory of staged static+dynamic partial analysis based on the theory of partial evaluation.
- Projections for efficiently generating the evaluators (and their generators) for partial results.
- Proved the correctness and precision of staged static+dynamic analysis.
- Calculated the percentage of callback affected methods and objects.

Conclusion

- Formalized the theory of staged static+dynamic partial analysis based on the theory of partial evaluation.
- Projections for efficiently generating the evaluators (and their generators) for partial results.
- Proved the correctness and precision of staged static+dynamic analysis.
- Calculated the percentage of callback affected methods and objects.

Thank You!