Call Graph Construction in Object-Oriented Languages

David Grove, Greg DeFouw, Jeffrey Dean*, and Craig Chambers

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, Washington 98195-2350 USA
{grove, gdefouw, jdean, chambers}@cs.washington.edu

Appeared in *OOPSLA '97 Conference Proceedings*

Presented by Long Cheng
09/16/2015

Background

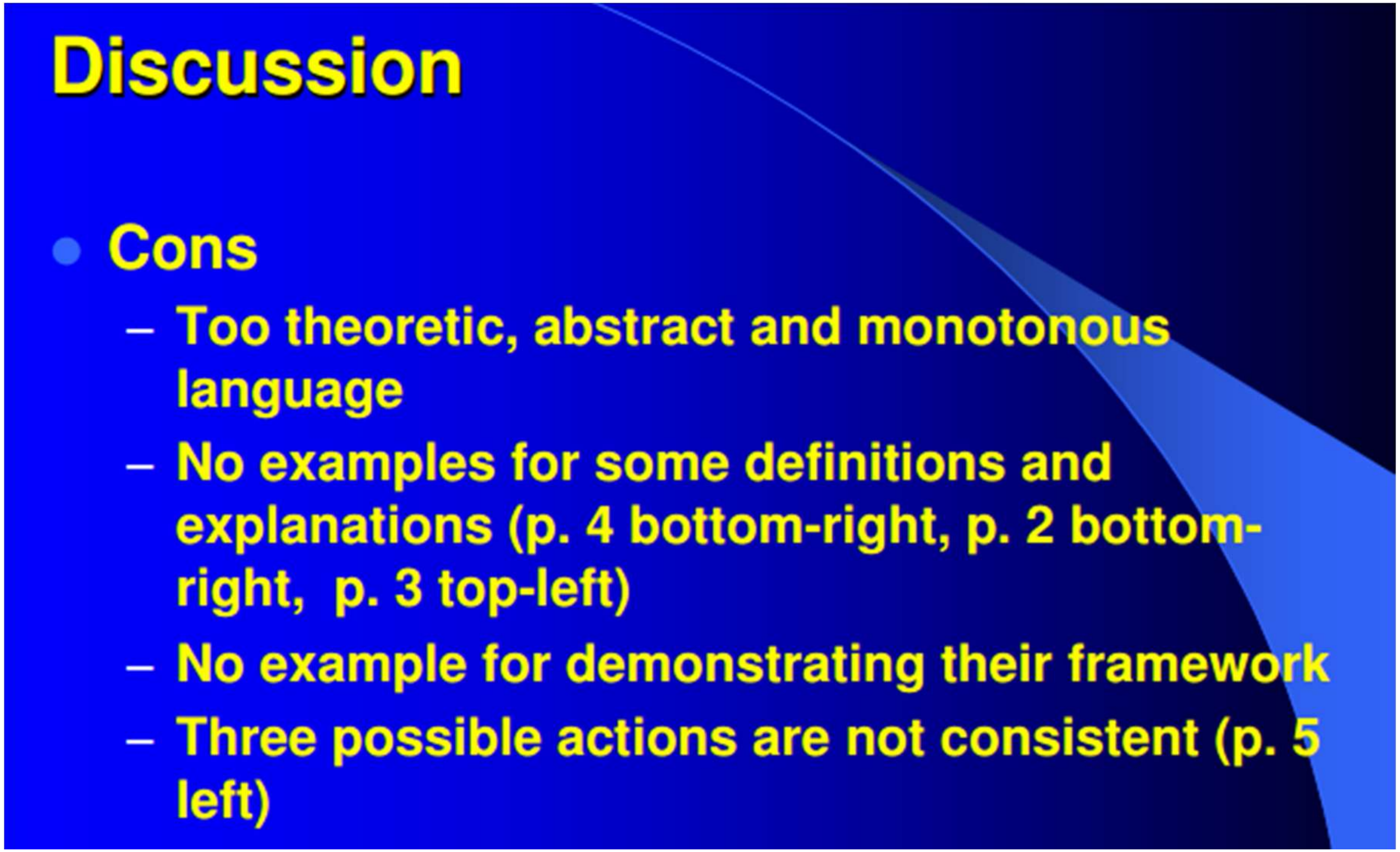
- Interprocedural analysis
  - calling relationships among procedures
  - optimize compilers to make less conservative assumptions across procedure call boundaries
  - enable substantial improvements in application performance

Motivation

- A number of call graph construction algorithms have been proposed
  - These algorithms make different **trade-offs** between the **precision** of the resulting call graph and any **associated dataflow information**, and the **cost** of computing the call graph
  - Lack of a general framework to express existing call graph construction algorithms

Main Contribution

- Develop a common framework for describing a wide range of existing call graph construction algorithms
  - Present a lattice-theoretic model of context-sensitive call graphs
    - element of the lattice <-> call graph for a program
- Survey existing algorithms
- Implement of the proposed framework and conduct empirical analysis of cost and benefit of algorithms



Source: http://www.ptidej.net/courses/ift6310/winter08/presentations2/080312/Presentation%20-%20Wei%20-%20Call%20Graph%20Construction%20in%20Object-Oriented%20Languages.pdf

Outline

- Modelling Call Graphs
  - Informal Model of Call Graphs
  - Formal (Lattice-Theoretic) Model of Call Graphs
- Generalized Call Graph Construction
- Experimental Assessment

Informal Model of Call Graphs

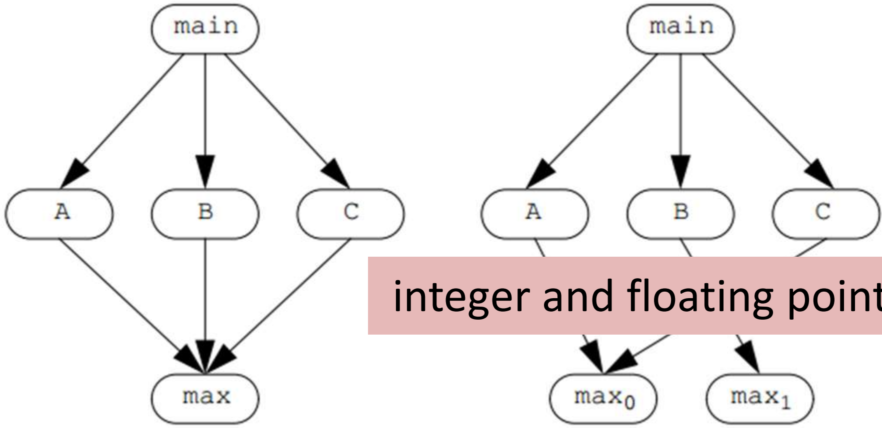
Each of these context-sensitive versions of a procedure is called a *contour.*

– a procedure may be analyzed separately for different calling contexts

```
procedure main() {
  return A() + B() + C();
}

procedure A() {
  return max(4, 7);
}
procedure B() {
  return max(4.5, 2.5);
}
procedure C() {
  return max(3, 1);
}
```

(a) Example Program



(b) Context-Insensitive (c) Context-Sensitive

integer and floating point parameters

Figure 1: Example Program and Call Graphs

Informal Model of Call Graphs

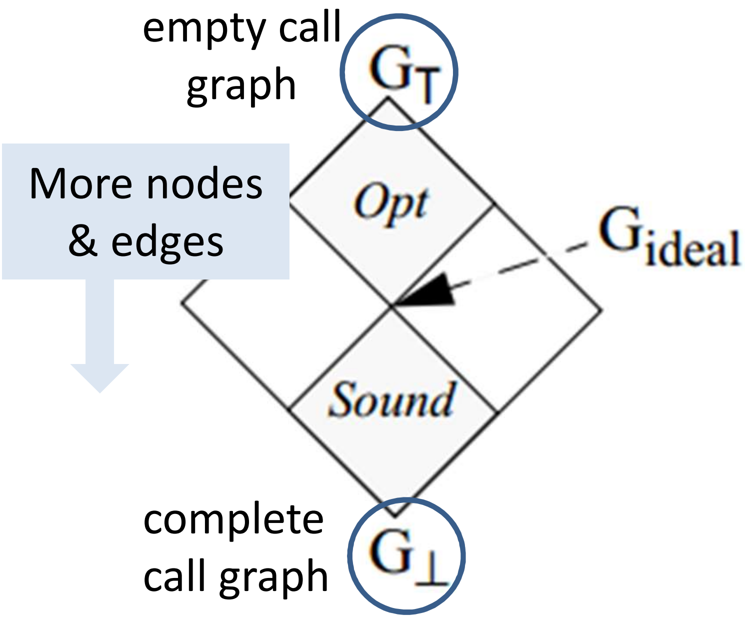
- What does a call graph include
  - Calling contour
  - Set of callee contour
  - Parameter class contour
  - Local variable contour
  - Procedure result contour

Informal Model of Call Graphs

- The different context-sensitive analyses differ in how they **determine what set of contours** to create for a given procedure and **which contours to select** as targets of a given call
- A wide range of context-sensitive call graphs can be represented by choosing different values for three parameterizing functions:
  - procedure contour selection function
  - instance variable contour selection function
  - class contour selection function

Formal Model of Call Graphs

- Use lattice-theoretic ideas to formally define the contour-based model of context-sensitive call graphs.

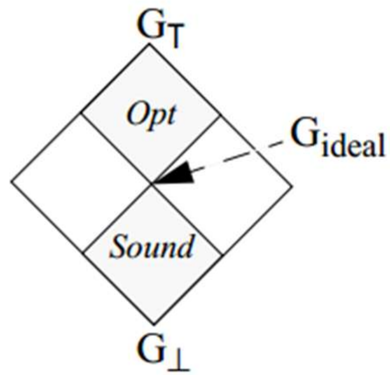


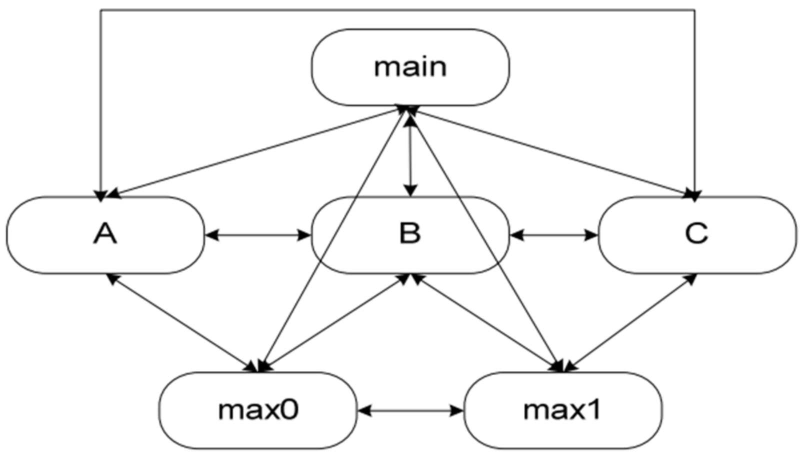
Regions in a Call Graph Lattice Domain

- Elements are call graphs
- One call graph below another if it is more conservative (less precise) than the other
- The point G_{ideal} identifies the "real" but usually uncomputable call graph, which can be described precisely as the greatest lower bound over all call graphs corresponding to actual program executions.

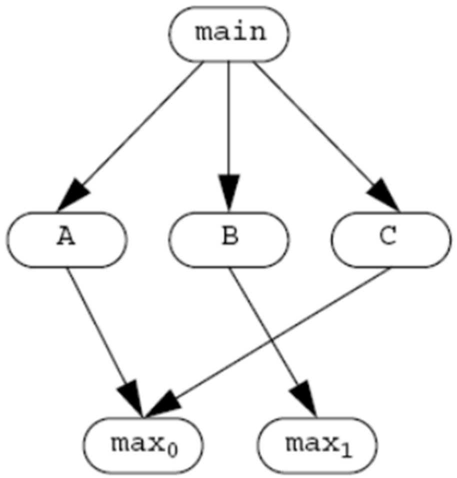
Formal Model of Call Graphs

- Lattice-Theoretic Model of Call Graphs





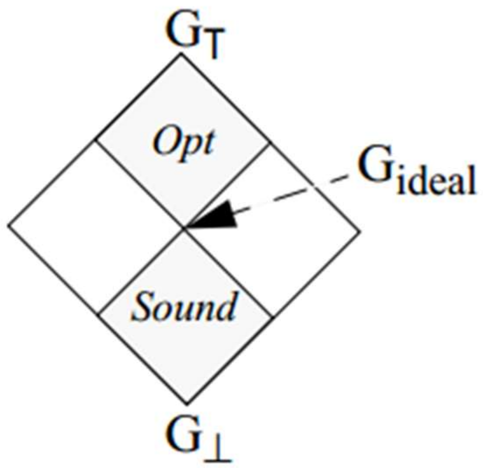
$G_\perp$: the complete call graph



G_{ideal} : real call graph

Formal Model of Call Graphs

- Soundness
  - A call graph is sound (i.e., safely approximates all possible program executions) if it is <u>at least as conservative as </u>each of the call graphs corresponding to possible program executions

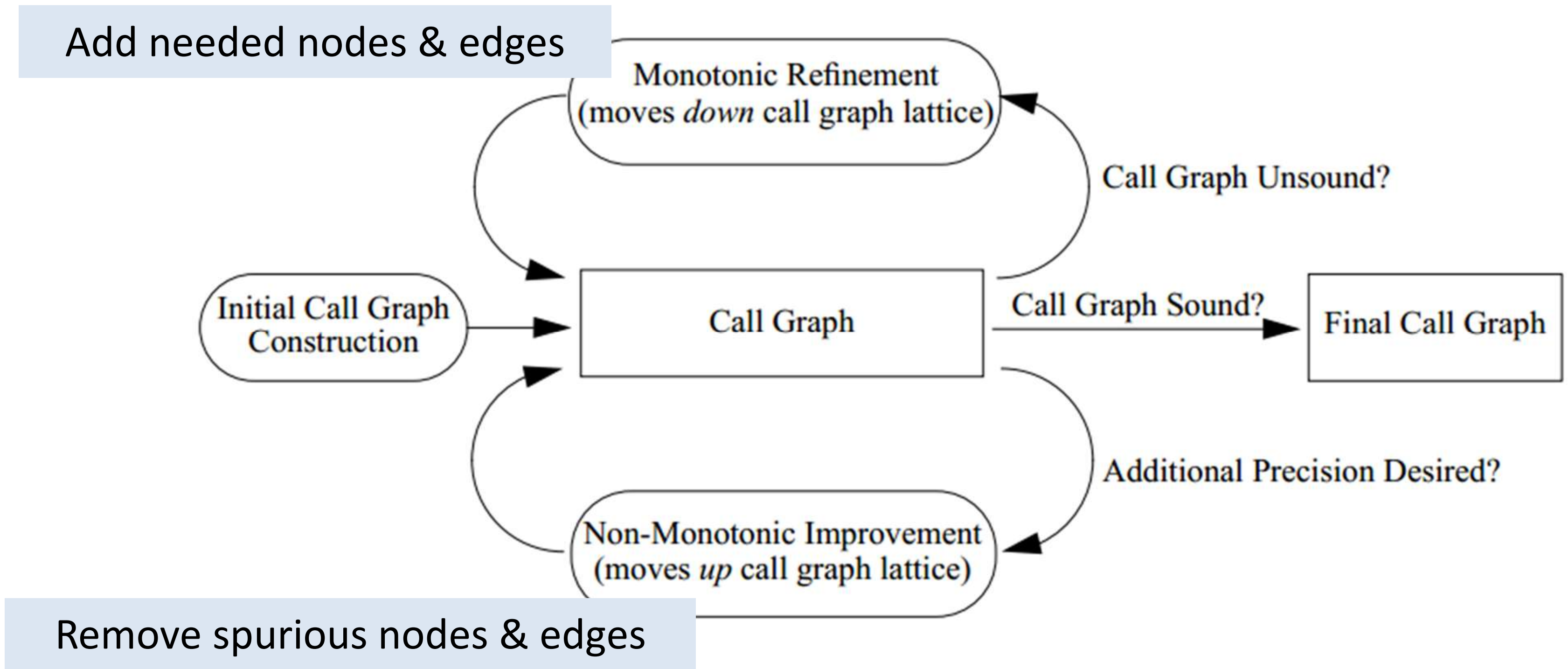


A ***sound*** call graph conservatively approximates the program's runtime behavior

- Every procedure called during some program execution is included
- Every call arc traversed during some program execution is included

Generalized Call Graph Construction

- Overview



Generalized Call Graph Construction Algorithm

Generalized Call Graph Construction

- Key parameters
  - The choice of domains for
    - ProcKey --- space of possible contexts for context-sensitive analysis of functions
    - InstVarKey --- space of possible contexts for separately tracking the contents of instance variables
    - ClassKey --- space of possible contexts for context-sensitive analysis of classes
  - The associated contour selection functions
  - The available non-monotonic improvement operations
  - Monotonic Refinement
  - Initial Call Graph

Generalized Call Graph Construction

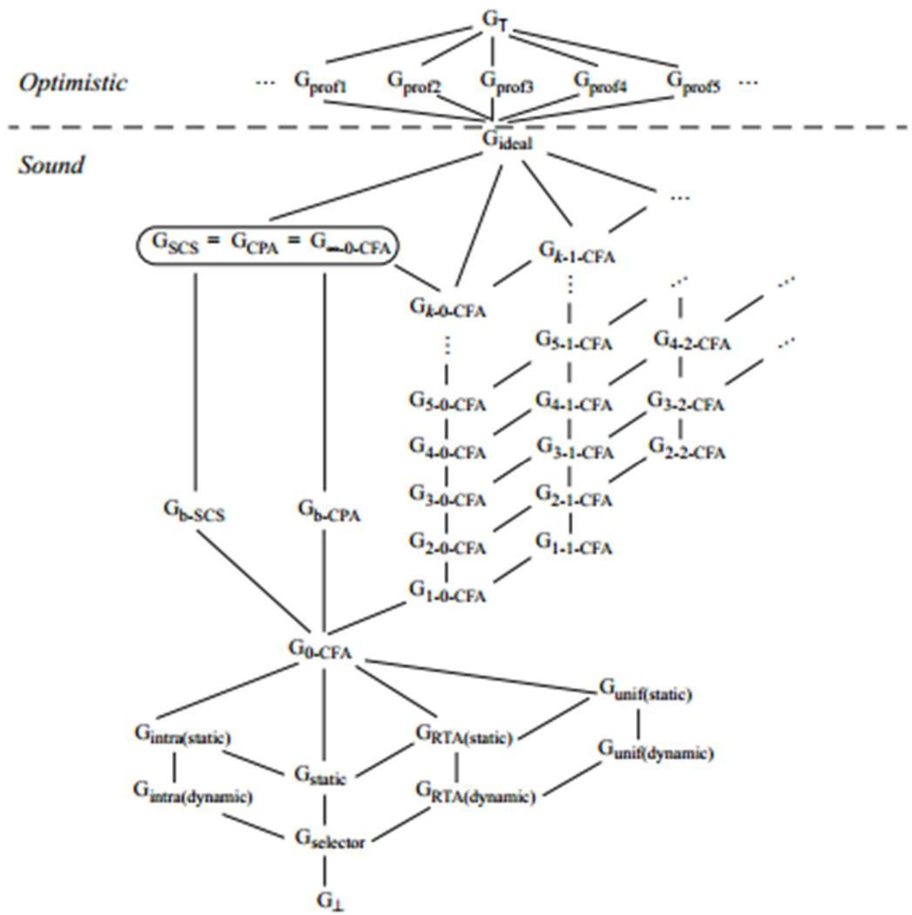
- Possible Initial Call Graphs
  - Although it is possible to use any element of the call graph lattice domain as an initial call graph, all existing algorithms start with one of two opposite extremes:
    - G_T: the top element of the call graph lattice (e.g., the empty call graph)
    - $G_\perp$: the bottom element of the call graph lattice (e.g., the complete call graph)

Generalized Call Graph Construction

- Possible Initial Call Graphs
  - G_T: the top element of the call graph lattice (e.g., the empty call graph)
    - Nodes/edges must be added
    - Potential for more precise final call graph
  - $G_\perp$: the bottom element of the call graph lattice (e.g., the complete call graph)
    - No further work required
    - May be very imprecise (especially with first-class functions)
    - Some (near-)linear-time algorithms:
      - Flow-insensitive: Bacon & Sweeney's Rapid Type Analysis (RTA) algorithm Steensgaard's near-linear-time points-to analysis
      - Limited flow-sensitive: DeFouw, Grove & Chambers's k-limited family of algorithms [POPL '98]

Generalized Call Graph Construction

- Relative Algorithmic Precision



The relative precision of the final products of the various call graph construction algorithms

Generalized Call Graph Construction

- Instantiating Call Graph Construction Framework
  - To turn framework into specific algorithm:
    - Choose an initial call graph construction method
    - Choose a contour selection function (e.g., 0-CFA, 1-CFA, CPA, SCS…)
    - Choose a spurious node/edge removal method (optional)

Experimental Assessment

- Framework has been implemented in Vortex optimizing compiler
  - 4,000 lines of shared code
  - 100-300 additional lines per algorithm
  - removing spurious node/edge component not implemented (non-monotonic improvement were under construction)

Experimental Assessment

- Goal: Evaluate costs and benefits on sizeable applications
- What are the costs of different call graph construction algorithms?
  - Analysis time
  - Analysis space
- What are the benefits of the resulting call graphs?
  - Call graph precision
  - Speed-up, resulting from interprocedural optimizations
  - Compiled code space, resulting from removing unreachable methods
- How practical is interprocedural analysis?

Experimental Assessment

- Benchmark Applications

6 Cecil programs
5 Java programs

	Program	Lines[a]	Description
Cecil Programs	richards	400	Operating systems simulation
	deltablue	650	Incremental constraint solver
	instr sched	2,400	Global instruction scheduler
	typechecker	20,000[b]	Typechecker for *old* Cecil type system
	new-tc	23,500[b]	Typechecker for *new* Cecil type system
	compiler	50,000	Old version of the Vortex optimizing compiler
Java Programs	toba	3,900	Java bytecode to C code translator
	java-cup	7,800	Parser generator
	espresso	13,800	Java source to bytecode translator[c]
	javac	25,550	Java source to bytecode translator[c]
	javadoc	28,950	Documentation generator for Java

- Analysis time for the flow-insensitive algorithms (G_{simple} and RTA) is linear in the size of the program
- k-l-CFA algorithms are time consuming
- In theory, SCS is worse than b-CPA, but the result of the experiment showed it is better
- Flow-sensitive algorithms are not suitable for large size programs.

• Cost … on Algorithms

6 algorithm families
(9 algorithms)

Analysis Time (secs),
Heap Space (MB),
Contours per Procedure,
Analyses per Procedure

	G_{simple}	RTA	0-CFA[b]	SCS	b-CPA	1-0-CFA	1-1-CFA	2-2-CFA	3-3-CFA
richards	2 sec 1.6 MB 1.0 / 1.0	2 sec 1.6 MB 1.0 / 1.0	3 sec 1.6 MB 1.2 / 2.2	3 sec 1.6 MB 1.8/ 2.0	4 sec 1.6 MB 2.4 / 2.9	4 sec 1.6 MB 1.9 / 3.0	5 sec 1.6 MB 1.9 / 3.7	5 sec 1.6 MB 2.4/ 3.8	4 sec 1.6 MB 2.8 / 4.0
deltablue	2 sec 1.6 MB 1.0/ 1.0	2 sec 1.6 MB 1.0 / 1.0	5 sec 1.6 MB 1.4 / 2.4	7 sec 1.6 MB 3.75 / 4.25	8 sec 1.6 MB 4.8 / 5.7	6 sec 1.6 MB 2.5 / 4.0	6 sec 1.6 MB 2.5 / 4.0	8 sec 1.6 MB 3.6 / 6.1	10 sec 1.6 MB 5.0 / 8.2
instr sched	6 sec 2.5 MB 1.0 / 1.0	4 sec 2.5 MB 1.0 / 1.0	67 sec 5.7 MB 1.4 / 4.8	83 sec 9.6 MB 6.5 / 8.5	146 sec 14.8 MB 11.8 / 17.0	99 sec 9.6 MB 3.5 / 10.3	109 sec 9.6 MB 3.5 / 10.6	334 sec 9.6 MB 6.7 / 24.9	1,795 sec 21.0 MB 13.3 / 48.3
typechecker	26 sec 12.0 MB 1.0 / 1.0	25 sec 5.5 MB 1.0 / 1.0	947 sec 45.1 MB 1.2 / 4.6			13,254 sec 97.4 MB 8.7 / 31.4			
new-tc	28 sec 6.9 MB 1.0 / 1.0	29 sec 6.9 MB 1.0 / 1.0	1,193 sec 62.1 MB 1.2 / 4.9			9,942 sec 115.4 MB 8.4 / 27.0			
compiler	87 sec 0.2 MB 1.0 / 1.0	93 sec 22.4 MB 1.0 / 1.0	11,941 sec 202.1 MB 1.3 / 8.8						
toba	35 sec 9.4 MB 1.0 / 1.0	18 sec 7.7 MB 1.0 / 1.0	79 sec 19.8 MB 1.0 / 1.0	67 sec 23.9 MB 1.1 /1.3	75 sec 19.8 MB 1.3 / 1.4	116 sec 20.3 MB 2.0 / 2.6	1,174 sec 19.8 MB 1.9 / 3.7	8,636 sec 19.8 MB 3.8 / 6.1	
java-cup	80 sec 76.1 MB 1.0 / 1.0	89 sec 82.4 MB 1.0 / 1.0	116 sec 76.6 MB 1.0 / 1.2	112 sec 76.1 MB 1.2 / 1.5	124 sec 76.2 MB 1.4 / 1.6	145 sec 87.8 MB 2.2/ 3.1	2,086 sec 76.0 MB 2.1 / 5.7		
espresso	49 sec 5.0 MB 1.0 / 1.0	74 sec 5.0 MB 1.0 / 1.0	136 sec 11.4 MB 1.0 / 1.4	307 sec 20.0 MB 1.8 / 2.5	305 sec 19.2 MB 2.0 / 2.9	1,183 sec 30.6 MB 3.7 / 7.3	51,646 sec 28.8 MB 3.6 / 16.3		
javac	74 sec 27.6 MB 1.0 / 1.0	35 sec 27.4 MB 1.0 / 1.0	289 sec 27.4 MB 1.0 / 1.7	442 sec 27.8 MB 2.2 / 3.2	562 sec 27.5 MB 2.3 / 3.4	2,068 sec 60.1 MB 4.5 / 10.4			
javadoc	66 sec 19.4 MB 1.0 / 1.0	38 sec 19.7 MB 1.0 / 1.0	169 sec 27.4 MB 1.0 / 1.3	165 sec 20.1 MB 1.6 / 1.9	208 sec 19.7 MB 1.6 / 2.0	295 sec 20.4 MB 2.6 / 3.6	27,991 sec 19.9 MB 2.1 / 5.9		

Experimental Assessment

- Cost and Precision of Call Graph Construction Algorithms

Average Static/Dynamic Callee Procedures for call sitea

	G_{simple}	RTA	0-CFA	SCS	b-CPA	1-0-CFA	1-1-CFA	2-2-CFA	3-3-CFA
richards	7.4 / 3.4	6.7 / 3.3	1.2 / 1.9	1.2 / 1.9	1.2 / 1.9	1.2 / 1.9	1.2 / 1.9	1.2 / 1.9	1.2 / 1.9
deltablue	10.2 / 8.1	9.4 / 7.3	1.4 / 2.2	1.4 / 2.2	1.4 / 2.2	1.4 / 2.2	1.4 / 2.2	1.4 / 2.2	1.4 / 2.1
instr sched	22.4 / 24.7	16.0 / 15.8	1.7 / 3.4	1.5 / 3.0	1.5 / 3.0	1.6 / 3.4	1.6 / 3.4	1.5 / 3.1	1.5 / 3.0
typechecker	46.7 / 59.3	42.9 / 53.4	4.4 / 13.9			4.0 / 11.9			
new-tc	56.4 / 60.2	52.8 / 55.6	4.0 / 10.5			3.8 / 10.3			
compiler	71.3 / 23.2	68.1 / 17.6	10.0 / 7.0						
toba	2.4 / 9.8	1.3 / 5.9	1.1 / 2.6	1.1 / 2.6	1.1 / 2.6	1.1 / 2.6	1.0 / 1.8	1.0 / 1.7	
java-cup	3.2 / 10.9	2.2 / 6.9	1.1 / 2.6	1.1 / 2.6	1.1 / 2.6	1.1 / 2.6	1.0 / 2.1		
espresso	2.2 / 10.8	2.1 / 10.1	1.7 / 9.7	1.7 / 9.7	1.7 / 9.7	1.7 / 9.7	1.6 / 8.7		
javac	3.9 / 11.6	1.4 / 6.8	2.2 / 5.5	2.2 / 5.5	2.2 / 5.5	2.2 / 5.5			
javadoc	3.1 / 11.1	1.4 / 7.2	1.2 / 3.4	1.2 / 3.4	1.2 / 3.4	1.2 / 3.4	1.1 / 1.4		

a. Shaded cells correspond to configurations that either did not complete in 24 hours or exhausted available virtual memory (450MB).

- For most programs, the simple interprocedurally flow-insensitive algorithms, G_{simple} and RTA, produced little improvement in execution speed.
- For the Cecil programs, interprocedurally flowsensitive algorithms (0-CFA and better) provided a significant boost in performance. Context-sensitivity was less important.
- For the java programs, the improvements are modest.

- Over

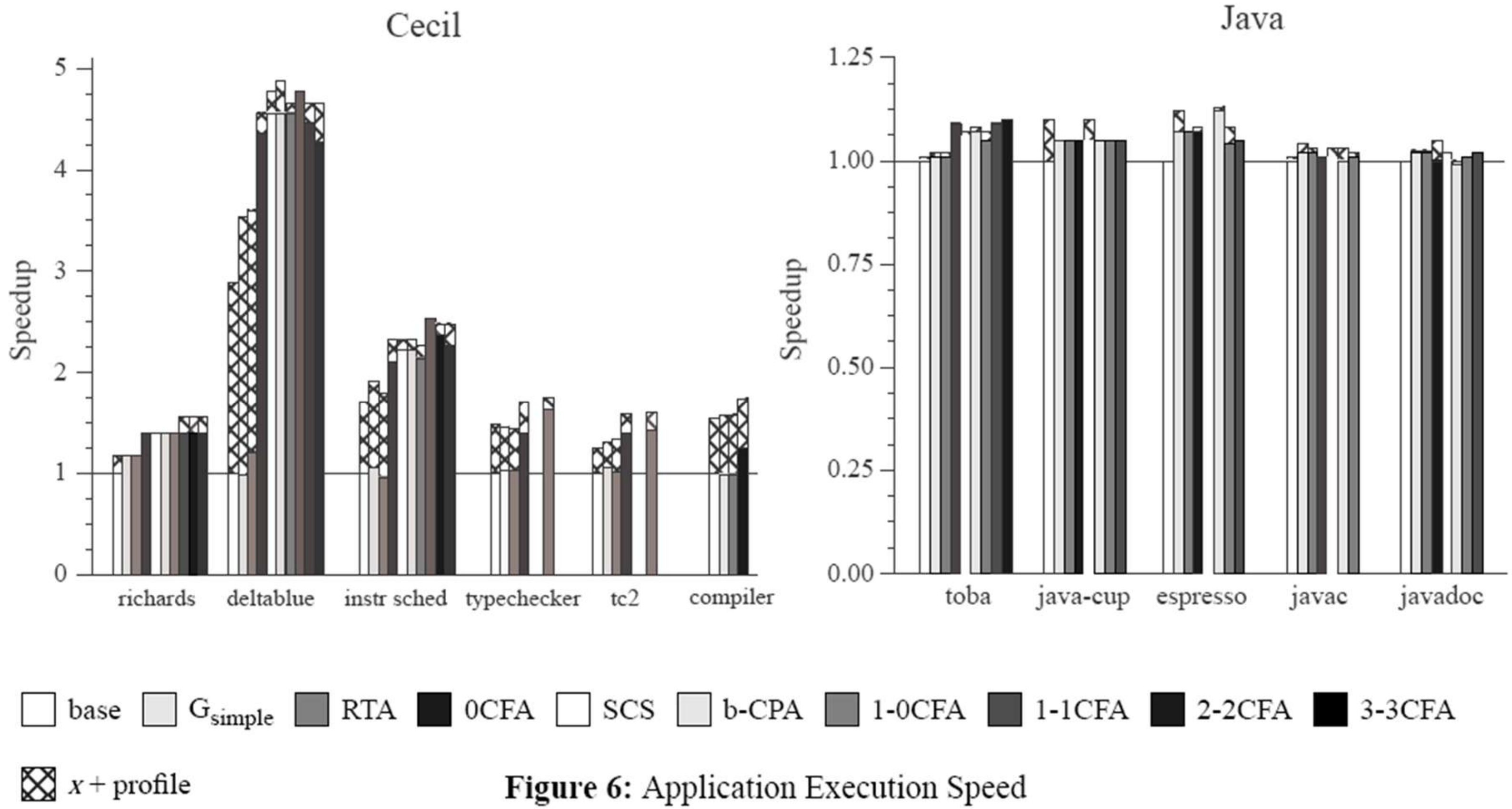


Figure 6: Application Execution Speed

Summary of Results

	Analysis Time	Speed-Up	Compiled Code Space
Cecil			
flow insensitive	◉	●	○
limited flow-sensitive	◓	◒	○
context-insensitive	◒	◉	◓
context-sensitive	●	◉	◓
Java			
flow insensitive	◉	●	○
limited flow-sensitive	◓	●	○
context-insensitive	◓	◒	○
context-sensitive	○	◒	○

Excellent ◉ ◓ ○ ◒ ● Poor

- Programming language/style impacts
  - Cecil
    - fast algorithms ineffective
    - can achieve large speed-up, but at a high cost
  - Java
    - analysis time reasonable, but speed-up small
- Scalability is a major concern for context-sensitive algorithms
- Even imprecise algorithms enable substantial code space reduction
  - Doing analysis actually reduces total compile time

Conclusion

- Unified model of call graph construction problem
- Experimental assessment using sizeable programs

- Future work
  - Extend algorithmic framework to better include linear-time algorithms
  - Investigating techniques to support incremental reconstruction of the program call graph and derive interprocedural information in the presence of program changes

New Development

- David Grove, Craig Chambers, “A Framework for Call Graph Construction Algorithms” ACM Transactions on Programming Languages and Systems, Vol. 23, No. 6, November 2001, Pages 685–746.
  - More formal lattice model
  - More examples!
  - New version of the framework
    - 9,500 lines of Cecil code
    - Support only monotonic algorithms
    - Wider range of algorithms
    - A scalable, near-linear-time algorithm

Acknowledgement

Some pictures are from the following materials:

https://static.aminer.org/pdf/PDF/000/522/227/call_graph_construction_in_object_oriented_languages.pdf

http://www.ptidej.net/courses/ift6310/winter08/presentations2/080312/Presentation%20-%20Wei%20-%20Call%20Graph%20Construction%20in%20Object-Oriented%20Languages.pdf

Thanks