Call Graph Construction in Object-Oriented Languages

David Grove, Greg DeFouw, Jeffrey Dean,
Craig Chambers

Course: IFT6310

Presented by Wei Wu
IRO, UdeM

2008-03-12

Outline

- Introduction
- Context
- Content
- Conclusion
- New development
- Related work
- Discussion
- References

Outline

- Introduction
  - Connection with Knuth
  - The authors
- Context
- Content
- Conclusion
- New development
- Related work
- Discussion
- References

Introduction

- “We understand complex things by systematically breaking them into successively simpler parts” (p.291 right)
- “but we’ve actually lost all the structure” (p.274 right)
- "No, the optimizing compiler would have to be so complicated (much more so than anything we have now) that it will in fact be unreliable." (p.282 right)
- "the compiler needs to be in a dialog with the programmer; it needs to know properties of the data, and whether certain cases can arise, etc." (p.283 left)
- “program manipulation system” (p.283 left)

Introduction

- David Grove
  - Research Staff Member, Watson Research Center (Hawthorne), IBM
  - Ph.D. in Computer Science from the University of Washington
  - Member of the Cecil/Vortex project
  - http://domino.watson.ibm.com/comm/research_people.nsf/pages/dgrove.index.html

Introduction

- Jeffrey Dean
  - Google Fellow in the Systems Infrastructure Group
  - Ph.D. in Computer Science from the University of Washington
  - http://research.google.com/people/jeff/

Introduction

- Greg DeFouw
  - Where is he ?
- Craig Chambers
  - Sounds familiar

Outline

- Introduction
- Context
  - What is Call Graph
  - What it for
- Content
- Conclusion
- New development
- Related work
- Discussion
- References

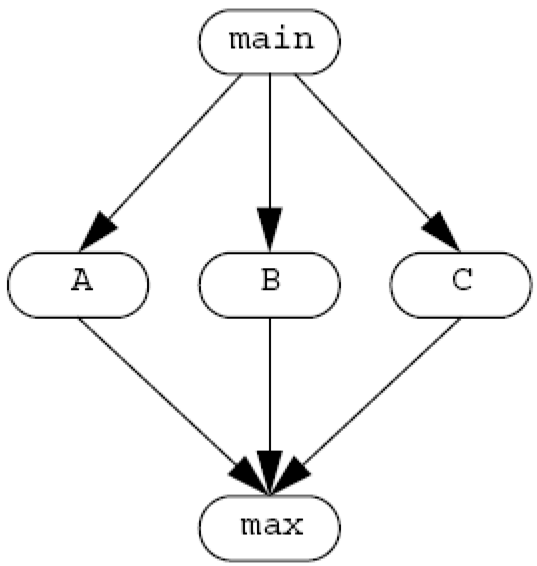
Context

- Call graph: a directed graph that represents the calling relationships between the program's procedures.

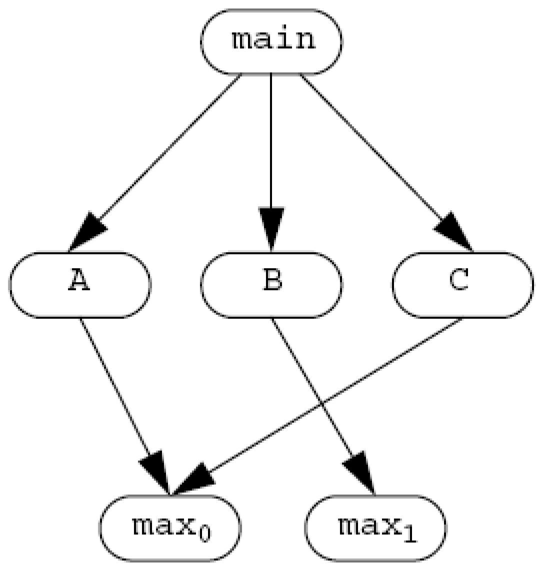
```
procedure main() {
  return A() + B() + C();
}

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

Context

- What it for?
  - human understanding of programs
  - Performance tuning
  - Design pattern detection
  - Other software maintenance activities, such as dead function detection, change impact analysis …

Outline

- Introduction
- Context
- Content
  - Informal Model of Call Graphs
  - Lattice-Theoretic Model of Call Graphs
  - Generalized Call Graph Construction Algorithm Framework
  - Implementation and experiment
- Conclusion
- New development
- Related work
- Discussion
- References

Content

- Informal Model of Call Graphs
  - Definition of Call Graph
  - Contour: a context-sensitive version of a procedure (object, class)

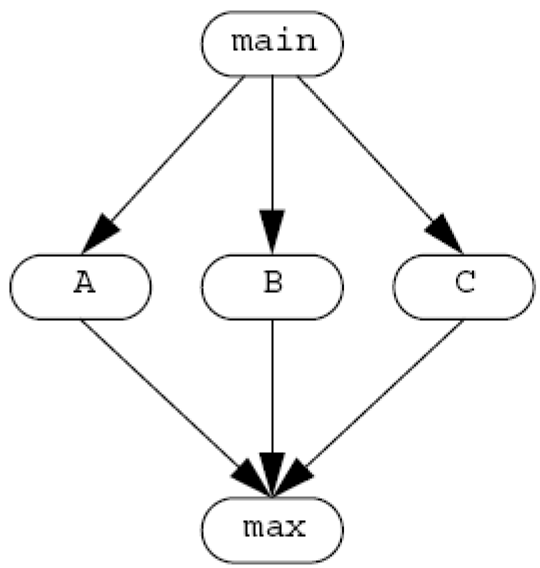
Content

- Informal Model of Call Graphs
  - Context-sensitive and Context-insensitive
  - Dynamic (profiling) static

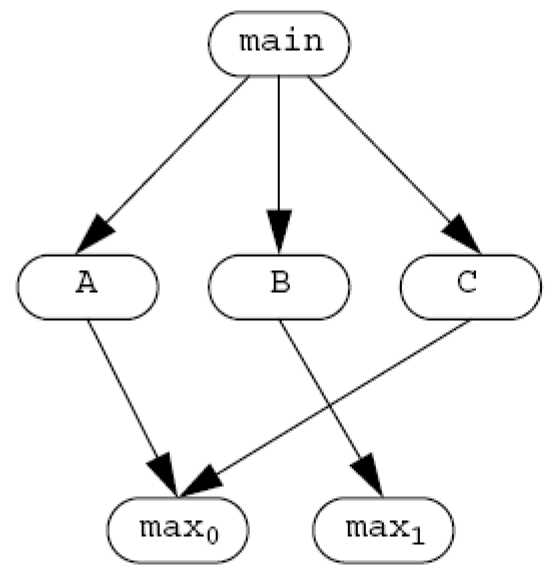
```
procedure main() {
  return A() + B() + C();
}

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

Content

- Informal Model of Call Graphs
  - What does a call graph include:
    - Calling contour
    - Set of callee contours
    - Parameter class contours
    - Local variable contours
    - Procedure result contour

Content

- **Informal Model of Call Graphs**
  - **Procedure contour select function**
    - **Input: Calling contour**

      ***<u>Possible</u>* classes of the actual parameters of the call**
    - **Output:** the set of callee contours

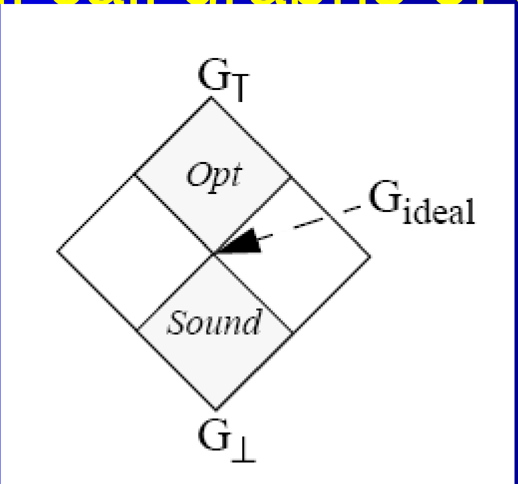
      The link between calling and callee contours

Content

- **Informal Model of Call Graphs**
  - **instance variable contour selection function**
    - **Input:** class set information about a variable
    - **Output:** appropriate instance variable contours
  - **class contour selection function**
    - **Input:** class instantiation site
    - **Output:** appropriate class contours

Content

- **Lattice-Theoretic Model of Call Graphs**
  - A mathematic description of the relation between all call graphs of a program.

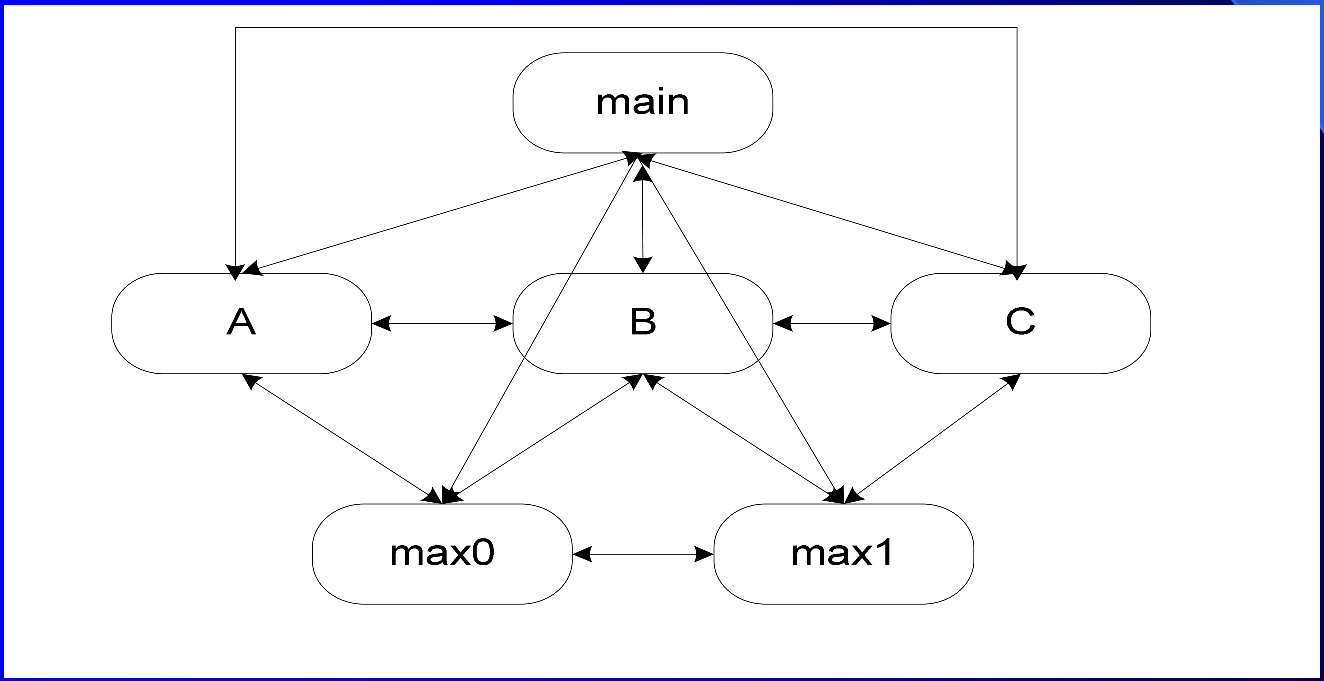


Content

- Lattice-Theoretic Model of Call Graphs
  - $G_{\top}$: empty call graph
  - $G_{\perp}$: complete call graph
  - G_{ideal} : real call graph

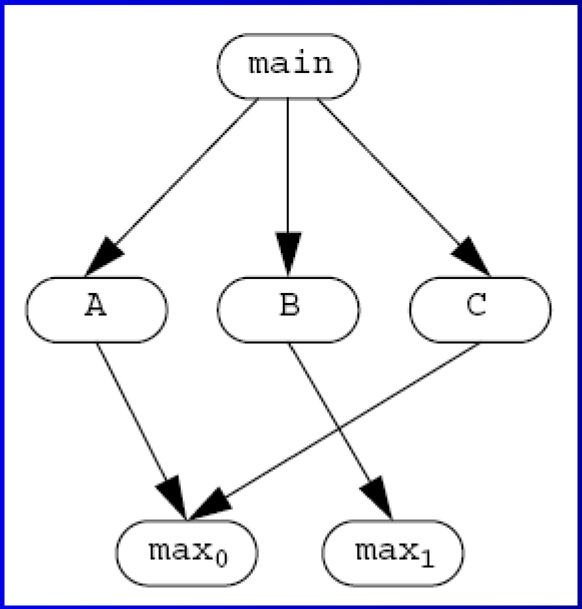
Content

- **Lattice-Theoretic Model of Call Graphs**
  - $G_{\perp}$ **: complete call graph**



Content

- **Lattice-Theoretic Model of Call Graphs**
  - G_{ideal} : real call graph



Content

- Lattice-Theoretic Model of Call Graphs
  - Mathematatic definitions of call graph

ClassContour	= *2Tuple(Class, ClassKey)*
ClassContourSet	= *Pow(ClassContour)*
InstVarContour	= *3Tuple(InstVariable, InstVarKey, ClassContourSet)*
InstVarContourSet	= *Pow(InstVarContour)*
ProcContour	= *7Tuple(Procedure, ProcKey, ProcContour, Map(Variable, ClassContourSet), Map(CallSite, ProcContourSet), Map(LoadSite, InstVarContourSet), Map(StoreSite, InstVarContourSet))*
ProcContourSet	= *Pow(ProcContour)*
CallGraph	= *2Tuple(ProcContourSet, InstVarContourSet)*

Figure 3: Definition of Call Graph Domain

Content

- **Generalized Call Graph Construction Algorithm Framework**

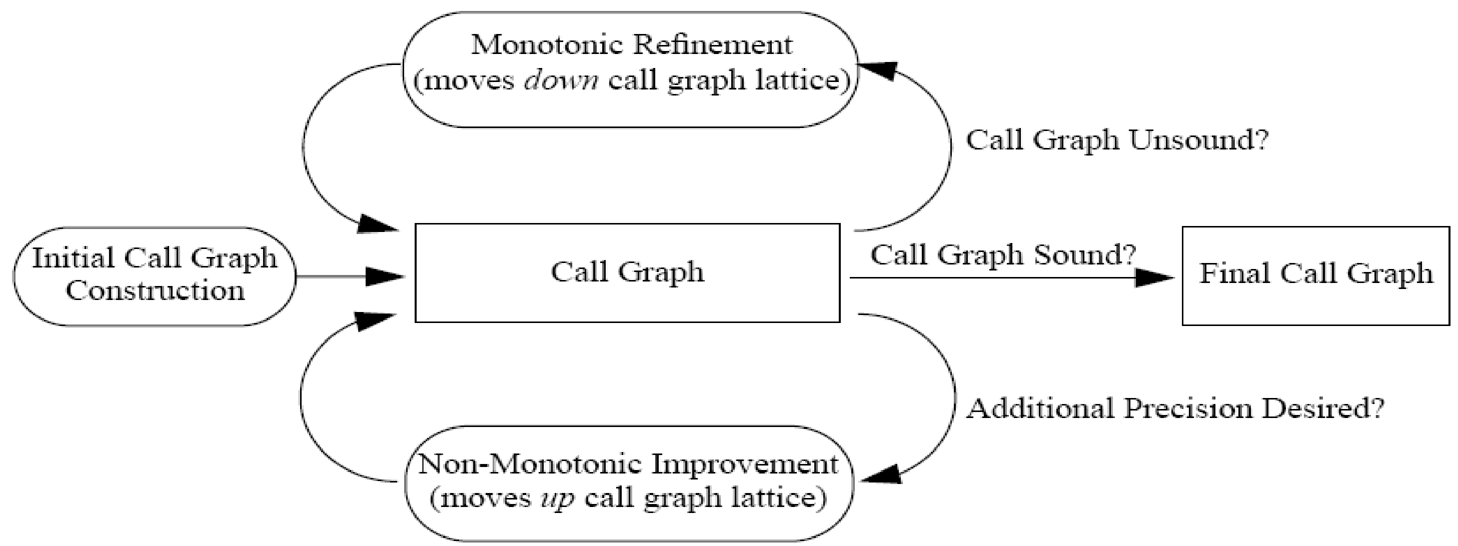


Figure 4: Generalized Call Graph Construction Algorithm

Content

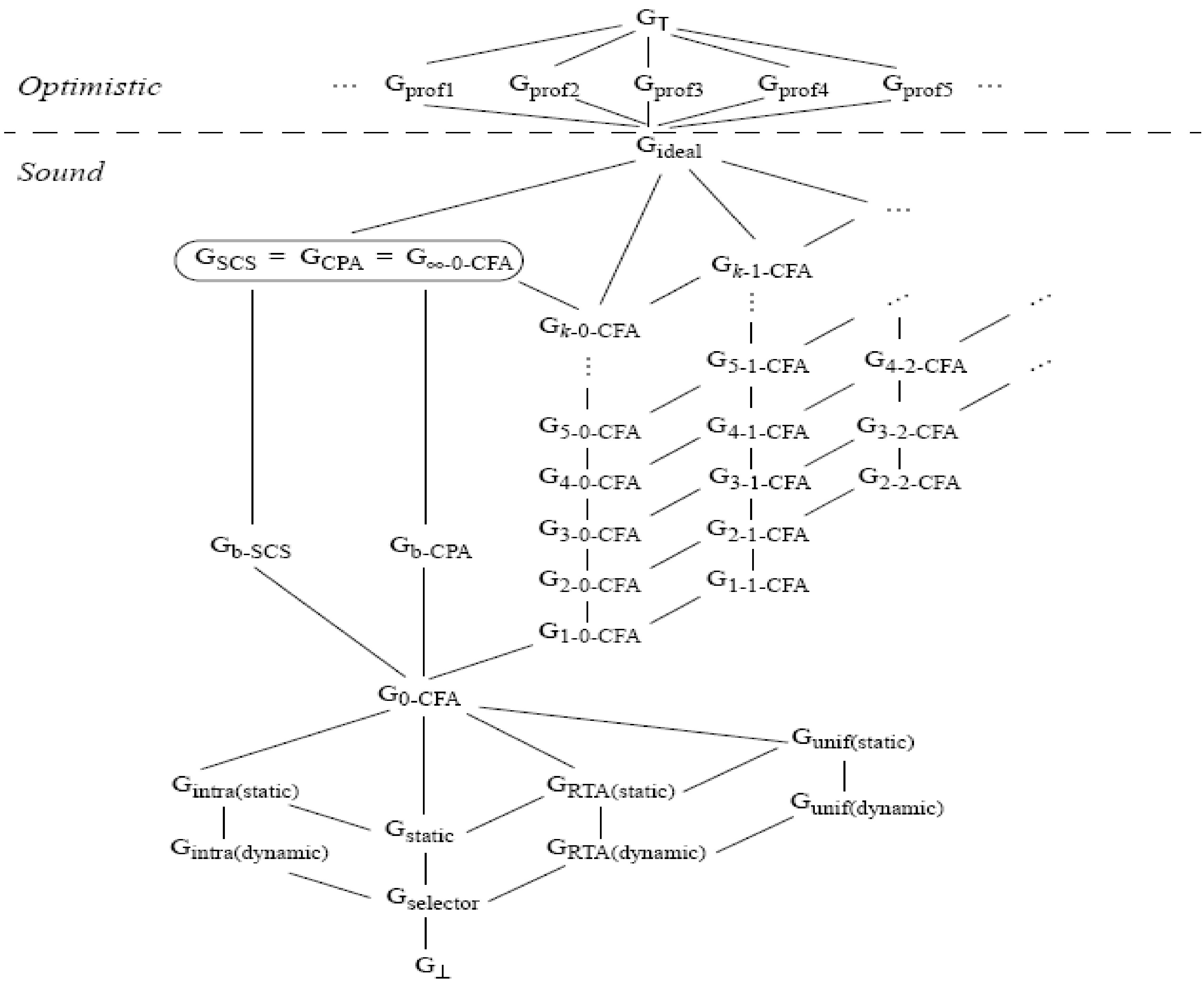
- Generalized Call Graph Construction Algorithm Framework
  - Key parameters
    - The choice of domains for ProcKey, InstVarKey, and ClassKey
    - The associated contour selection functions
    - The available non-monotonic improvement operations

Content

- Generalized Call Graph Construction Algorithm Framework
  - Other parameters
    - Initial Call Graph
    - Monotonic Refinement (same in all algorithms)

Content

- **Generalized Call Graph Construction Algorithm Framework**
  - **Relative algorithmic precision**
    - *Under the no-specialization assumption,* ∞-0-CFA, SCS, and CPA all produce call graphs with identical effective precision.



Optimistic
Sound
$G_{\top}$
G_{prof1}
G_{prof2}
G_{prof3}
G_{prof4}
G_{prof5}
G_{ideal}
$G_{SCS} = G_{CPA} = G_{\infty\text{-}0\text{-}CFA}$
$G_{k\text{-}1\text{-}CFA}$
$G_{k\text{-}0\text{-}CFA}$
$G_{5\text{-}1\text{-}CFA}$
$G_{4\text{-}2\text{-}CFA}$
$G_{5\text{-}0\text{-}CFA}$
$G_{4\text{-}1\text{-}CFA}$
$G_{3\text{-}2\text{-}CFA}$
$G_{4\text{-}0\text{-}CFA}$
$G_{3\text{-}1\text{-}CFA}$
$G_{2\text{-}2\text{-}CFA}$
$G_{3\text{-}0\text{-}CFA}$
$G_{2\text{-}1\text{-}CFA}$
$G_{b\text{-}SCS}$
$G_{b\text{-}CPA}$
$G_{2\text{-}0\text{-}CFA}$
$G_{1\text{-}1\text{-}CFA}$
$G_{1\text{-}0\text{-}CFA}$
$G_{0\text{-}CFA}$
$G_{unif(static)}$
$G_{intra(static)}$
$G_{RTA(static)}$
$G_{unif(dynamic)}$
G_{static}
$G_{intra(dynamic)}$
$G_{RTA(dynamic)}$
$G_{selector}$
$G_{\perp}$

Content

- **Implementation**
  - **Vortex optimizing compiler infrastructure**
    - 4,000 lines of Cecil code
    - contour and contour_key abstract classes and related data structures
    - centralized code for executing the generalized algorithm and monotonic refinement

Content

- **Implementation**
  - **Vortex optimizing compiler infrastructure**
    - ipca_algorithm abstract class that defines three contour selection functions
    - Abstract mix-in classes for managing procedure, instance variable, and class contours
    - non-monotonic improvement were under construction

Content

- Implementation
  - Time/Space Tradeoffs
    - Eagerly approximating class sets during set union operations
    - Three largest Cecil programs, 0-CFA

      Analysis time is reduction: a factor of 15

      The resulting optimized executables slowdown: 2% to 8%

Content

- **Implementation**
  - **Sparse procedure representation**
    - Remove details of non-object data and control flow
    - For several of the smaller Java programs: Analysis time and memory usage reduction: 50%
    - Implementation was not complete

Content

- Experiment
  - 6 algorithm families (9 algorithms)
  - 6 Cecil programs
  - 5 Java programs
  - Compare:
    - Precision
    - Cost
    - Execution speed and size

Content

- Experiment

Table 2: Benchmark Applications

	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

	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		

Content

- Experiment
  - Analysis time for the flow-insensitive algorithms (Gsimple 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

Content

- Experiment
  - ***Flow-sensitive*** algorithms are not suitable for large size programs.
  - the context-sensitive algorithms did not provide much additional precision over the ***context-insensitive*** 0-CFA algorithm.
  - The authors expected ***flow-sensitivity*** (0-CFA) to provide the main improvements in bottom-line execution speed, with ***flow-insensitive*** algorithms much worse and ***context-sensitive*** algorithms not much better.

Content

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

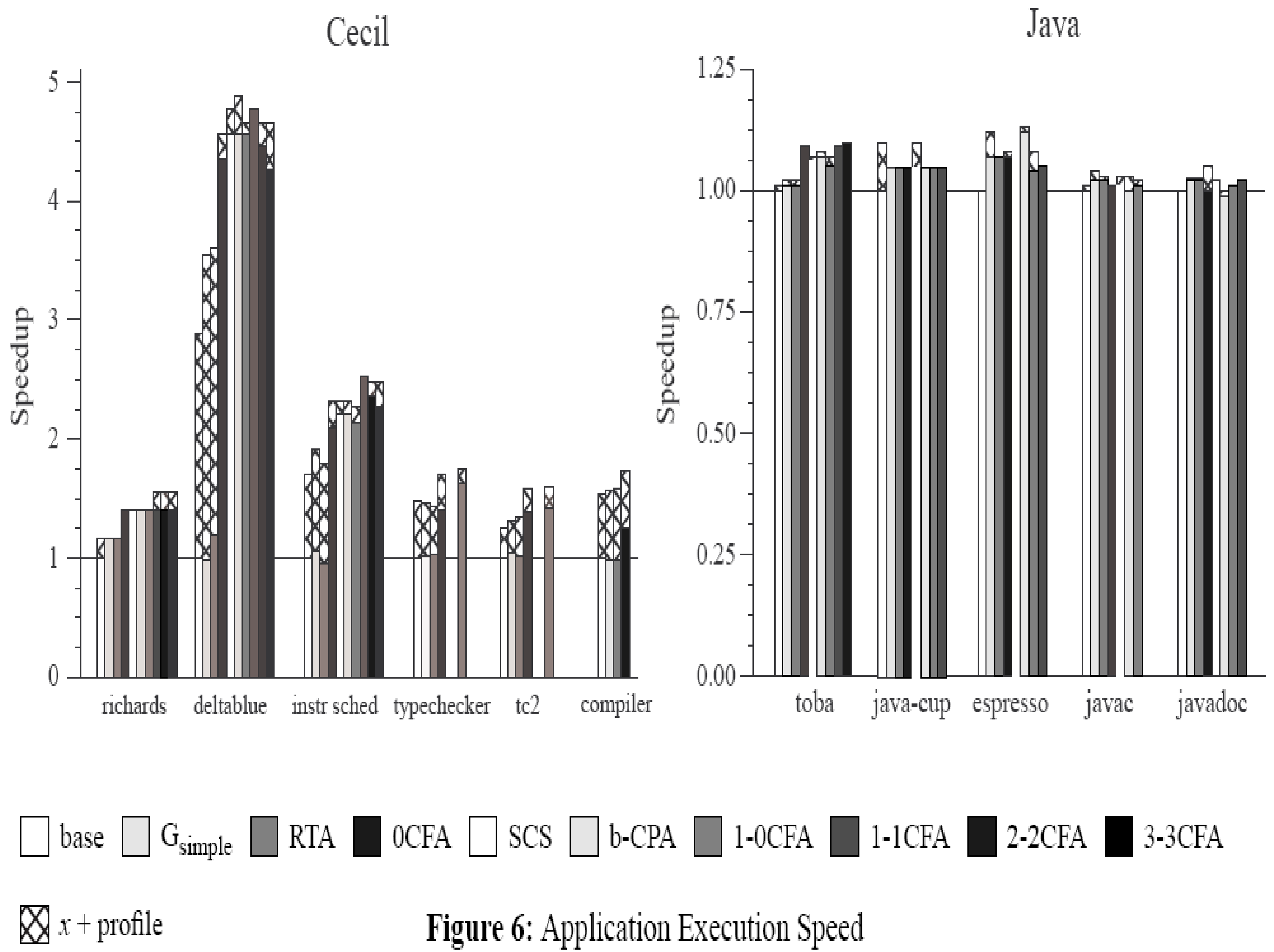


Figure 6: Application Execution Speed

Content

- Executable code size comparison
  - The treeshaking optimization reduced executable sizes for all the interprocedural analysis configurations.
  - For the Java programs, reductions were 10% to 20%. The flow-sensitive algorithms' reductions were 0% to 3% more than the flow-insensitive.
  - For the Cecil programs, reductions were 15% to 40% . The Interprocedurally flow-sensitive algorithms brought additional 10% over the flow-insensitive algorithms.
  - Context-sensitive call graphs did not measurably improve the effectiveness

Outline

- Introduction
- Context
- Content
- Conclusion
- New development
- Related work
- Discussion
- References

Conclusion

- Interprocedural analyses, especially interprocedural class analysis, enabled substantial speedups for Cecil programs but only modest speedups for Java programs.
- The call graphs constructed by the interprocedurally flow-sensitive algorithms had a large impact on the effectiveness of client interprocedural analyses and subsequent optimizations.
- The influences of the more precise call graphs constructed by the context-sensitive algorithms are not more significant.

Outline

- Introduction
- Context
- Content
- Conclusion
- New development
- Related work
- Discussion
- References

New development

- 2001, David Grove, Craig Chambers
  - «A Framework for Call Graph Construction Algorithms»
  - 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

Outline

- Introduction
- Context
- Content
- Conclusion
- New development
- Related work
- Discussion
- References

Related work

- Susan L. Graham, Peter B. Kessler, Marshall K. Mckusick, Gprof: A call graph execution profiler, 1982
- Linda Badri, Mourad Badri and Daniel St-Yves, Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique, 2005
- Bohnet, J and Dollner, J., Facilitating Exploration of Unfamiliar Source Code by Providing 2½D Visualizations of Dynamic Call Graphs
- Weilei Zhang, Barbara Ryder, Constructing Accurate Application Call Graphs For Java To Model Library Callbacks, 2006

Related work

- Intel® VTune™ Performance Analyzer
- Gprof
- Python call graph
- Egypt (for C)

Outline

- Introduction
- Context
- Content
- Conclusion
- New development
- Related work
- <u>Discussion</u>
- References

Discussion

- Pros
  - Using a uniform framework eases analyzing and comparing different call graph construction algorithms
  - Survey existing algorithms
  - Implementation of their framework
  - Empirical analysis

Discussion

- Cons
  - Too theoretic, abstract and monotonous language
  - No examples for some definitions and explanations (p. 4 bottom-right, p. 2 bottom-right, p. 3 top-left)
  - No example for demonstrating their framework
  - Three possible actions are not consistent (p. 5 left)

References

- 1. David Grove and Craig Chambers, A framework for call graph construction algorithms, ACM Transactions on Programming Languages and Systems, Volume 23 , Issue 6 (November 2001), Pages: 685 - 746, 2001, ACM
- 2. Susan L. Graham, Peter B. Kessler and Marshall K. Mckusick, Gprof: A call graph execution profiler, ACM SIGPLAN Notices, Volume 17 , Issue 6 (June 1982), Proceedings of the 1982 SIGPLAN symposium on Compiler construction, Pages: 120 - 126, 1982, ACM
- 3. Linda Badri, Mourad Badri and Daniel St-Yves, Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique, Proceedings of the 12th Asia-Pacific Software Engineering Conference, Pages: 167 - 175, 2005, IEEE Computer Society
- 4. Weilei Zhang, Barbara Ryder, Constructing Accurate Application Call Graphs For Java To Model Library Callbacks, SCAM; Vol. 6, Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, Pages: 63 - 74, 2006, IEEE Computer Society
- 5. Bohnet, J and Dollner, J., Facilitating Exploration of Unfamiliar Source Code by Providing 2½D Visualizations of Dynamic Call Graphs, Visualizing Software for Understanding and Analysis, VISSOFT 2007. 4th IEEE International Workshop, June 2007,Pages: 63-66

Thank you!