Using Annotations to Speed Up Run-time Optimization

Ondřej Lhoták

November 26, 2001

308-762

References

- C. Krintz, B. Calder. *Using Annotations to Reduce Dynamic Optimization Time.* PLDI '01
- P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, C. Verbrugge. *A Framework for Optimizing Java Using Attributes.* CC '01
- J. Jones, S. Kamin. *Annotating Java Class Files with Virtual Registers for Preformance.* Concurrency:P&E 12(6) p. 389 May '00
- A. Azevedo, A. Nicolau, J. Hummel. *Java Annotation-Aware Just-In-Time (AJIT) Compilation System.* Java Grande '99

Introduction

- Compiler analyses are slow
- JIT compilers need to be fast
- Use annotations to convey analysis results to JIT
- Can also encode profiling information

Issues

- What information is useful?
  - Overall performance
  - Startup time
- Verifiability/security
- How to encode annotations?
  - Flexibility
  - Annotation size

List of Annotations

- Virtual register allocation (ANH, JK, KC)
- Basic-block graph (KC)
- Intermediate representation reuse (KC)
- Hotness of method (KC)
- Applicability of optimization (KC)
- Array bounds check elim. (Sable, ANH)
- Null pointer check elim. (Soot, ANH)

Virtual Register Allocation

- Assign a virtual register to every value
- Prioritize virtual registers
- (JK) Add move/swap annotations
- (ANH) Aggressively optimize RTL, then translate back to heavily annotated bytecode
- Annotation size: 31% - 97% of bytecode

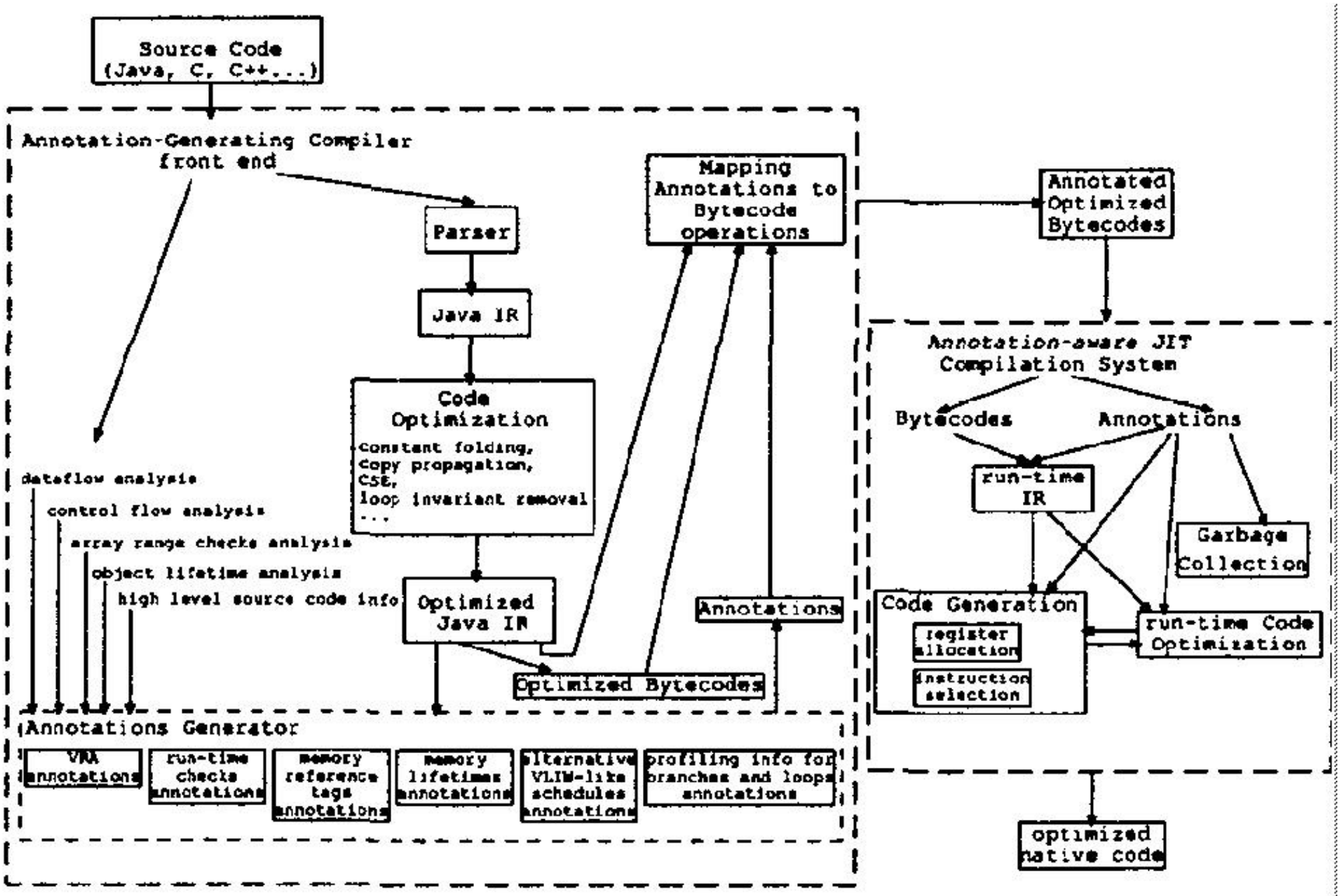
Virtual Register Allocation

Case 1: Array element address calculation and array load

Bytecode	Java IR
iaload	V0 holds array address V1 holds index 1 : ishl V1, "ishift", V2 2 : iadd V2, "arraySizeOffset", V2 3 : aadd V0, V2, V3 4 : ild (V3), V4

Annotated Bytecode

opcode	SRC	SRC	EXTRA	EXTRA	DEST
iaload	V0	V1	V2	V3	V4


Source Code
(Java, C, C++...)
Annotation-Generating Compiler
front end
Parser
Java IR
Code
Optimization
constant folding,
copy propagation,
CSE,
loop invariant removal
...
Optimized
Java IR
Mapping
Annotations to
Bytecode
operations
Annotated
Optimized
Bytecodes
dataflow analysis
control flow analysis
array range checks analysis
object lifetime analysis
high level source code info
Annotations
Optimized Bytecodes
Annotations Generator
VRA
annotations
run-time
checks
annotations
memory
reference
tags
annotations
memory
lifetimes
annotations
alternative
VLIW-like
schedules
annotations
profiling info for
branches and loops
annotations
Annotation-aware JIT
Compilation System
Bytecodes
Annotations
run-time
IR
Garbage
Collection
Code Generation
register
allocation
instruction
selection
run-time Code
Optimization
optimized
native code

Speedup

Benchmarks	SUN Interpreter (in secs)	kaffe JIT (in secs)	AJIT (in secs)
Neighbor 256X256 array Iterations = 1500	553.03	162.73	115.31
EM3D 1250 tree nodes Iterations = 200	359.84	149.86	74.51
Bitonic Sort 1024 tree nodes Iterations = 512	167.05	141.23	120.96
Huffman 30000 array nodes Iterations = 288	4690.00	1856.00	1487.00

Virtual Register Allocation

- (KC) Statically count number of uses of each variable
- Let ORP JIT compiler do the actual register allocation
- Annotation size: 0.5% of bytecode

Basic Block Graph

- Put adjacency-list representation of BB graph in annotation to save the compiler multiple passes over the bytecode
- Annotation size: 5.2% of bytecode

Intermediate Rep. Reuse

- Example: inlining
- ORP preprocesses methods prior to inlining them into callers
- Save preprocessed method, or not?
- Annotation size: 0.0% of bytecode

Hotness of method

- One bit per method: O1 or O3
- Profiling determined that the 25% most frequently executed methods are worth optimizing
- Annotation size: 0.3% of bytecode

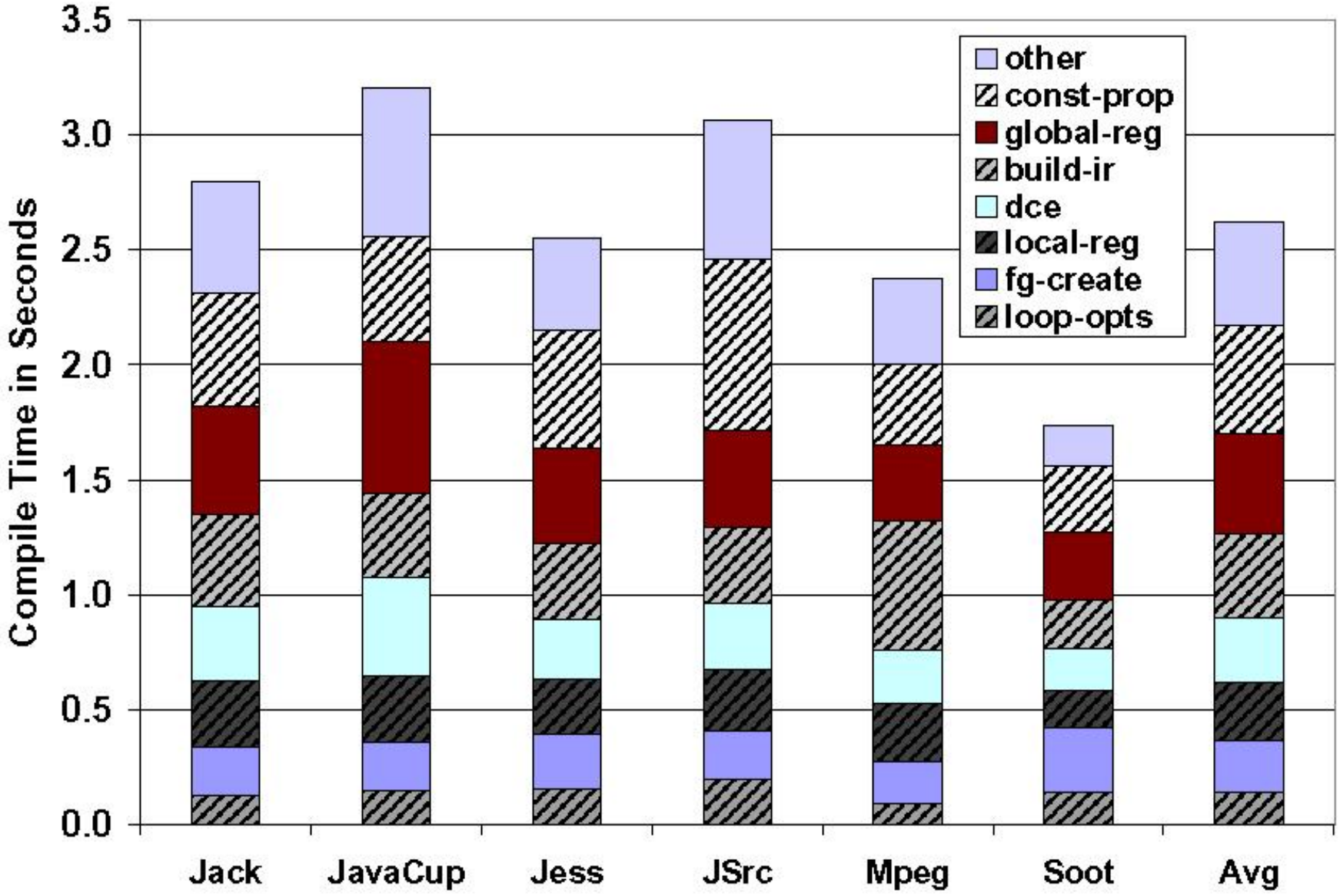
Applicability of optimization

- Example: profiling determined that constant propagation only benefits 70% of the methods
- One bit per method: do or don't do CP
- Can use for other optimizations
- Annotation size: 0.0% of bytecode

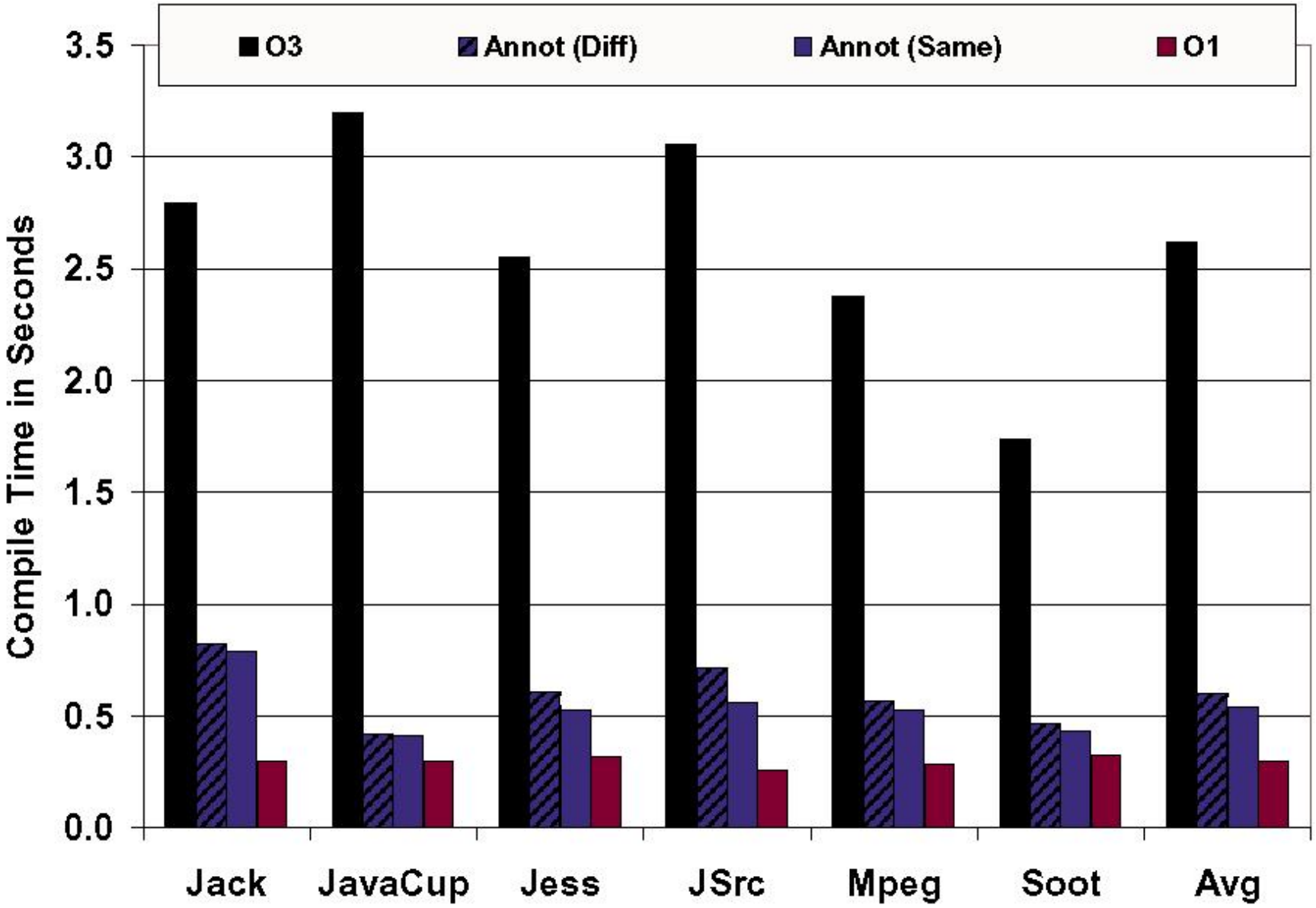
Benchmarks

Program	Cmpl Time in secs		Total Time(s) in secs	
	O1	O3	O1	O3
Jack	0.30	2.80	42.2	41.5
JavaCup	0.30	3.20	48.9	47.7
Jess	0.32	2.55	44.1	42.9
JSrc	0.26	3.06	49.6	48.2
Mpeg	0.28	2.37	37.8	31.2
Soot	0.33	1.74	7.0	6.9
Avg	0.30	2.62	38.3	36.4

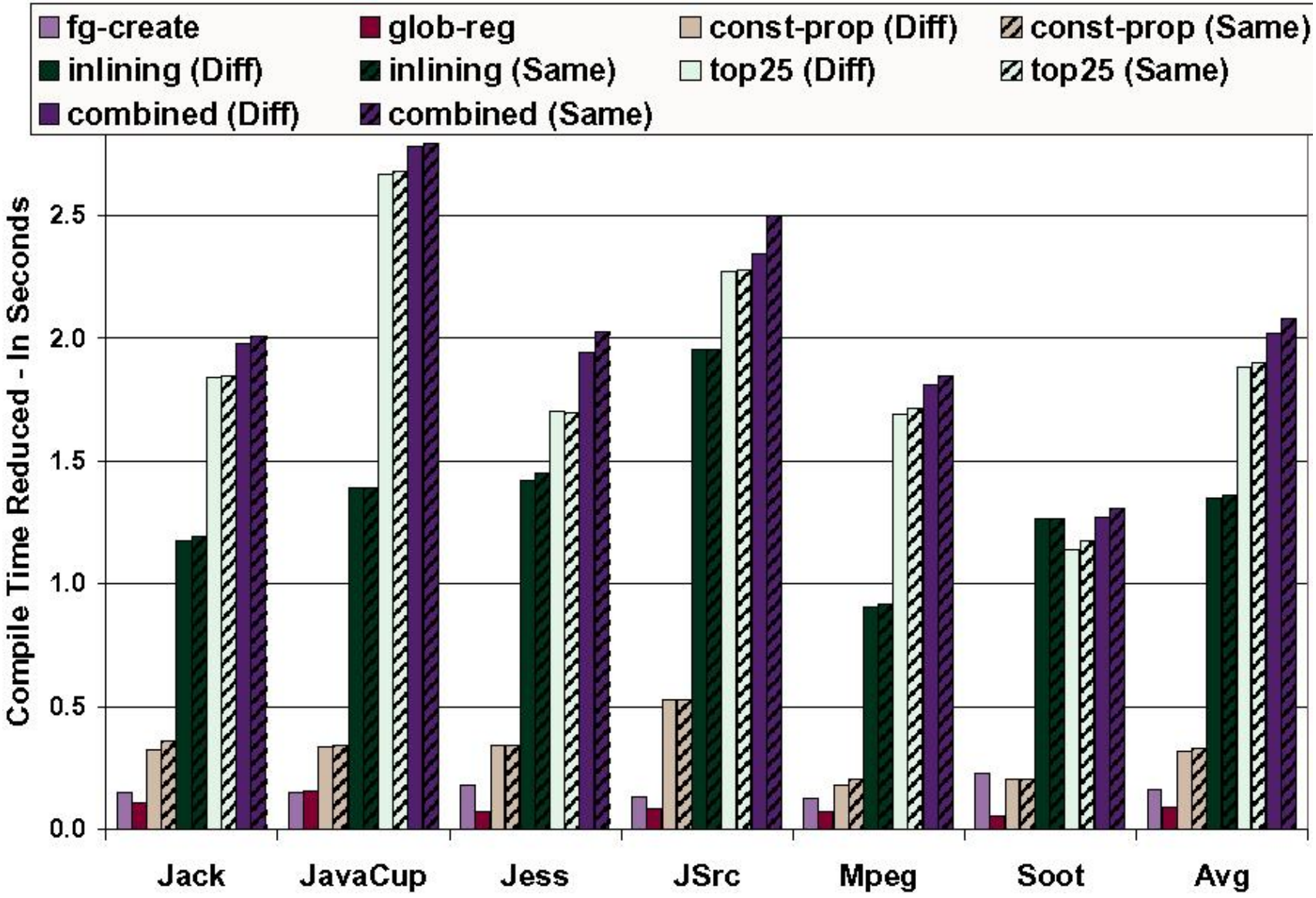
Benchmarks


Compile Time in Seconds
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
other
const-prop
global-reg
build-ir
dce
local-reg
fg-create
loop-opts
Jack
JavaCup
Jess
JSrc
Mpeg
Soot
Avg

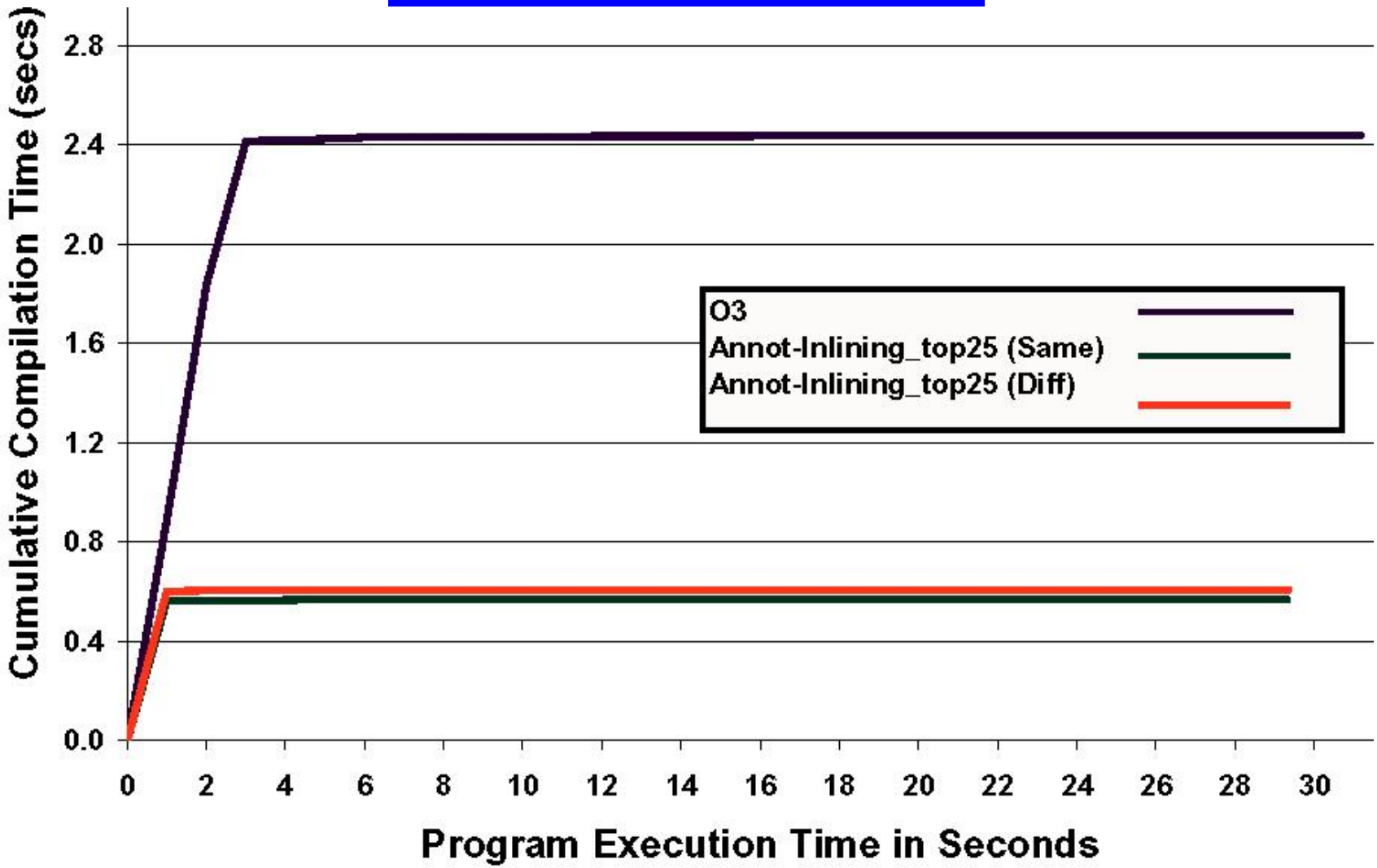
Speedup


O3
Annot (Diff)
Annot (Same)
O1
Compile Time in Seconds
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Jack
JavaCup
Jess
JSrc
Mpeg
Soot
Avg

Speedup


fg-create
glob-reg
const-prop (Diff)
const-prop (Same)
inlining (Diff)
inlining (Same)
top25 (Diff)
top25 (Same)
combined (Diff)
combined (Same)
Compile Time Reduced - In Seconds
0.0
0.5
1.0
1.5
2.0
2.5
Jack
JavaCup
Jess
JSrc
Mpeg
Soot
Avg

Startup Time



Mpeg

Security

- Unsafe:
  - Virtual register allocation (ANH, JK)
  - Array bounds check elimination (Sable)
  - Null pointer check elimination (Sable)
  - Basic-block graph (KC)
- Safe:
  - Virtual register allocation (KC)
  - Intermediate representation reuse (KC)
  - Hotness of method (KC)
  - Applicability of optimization (KC)

Soot Annotation Format

- Attributes for class, field, method
- Statement annotations aggregated in method attribute
- Nice interface in Soot for creating annotations
- No effort to compress attributes

(KC) Annotation Format

- All annotations aggregated into single class attribute with one-character name
  - Annotation opcode (2 bytes)
  - Size (2 bytes)
  - Method id (optional) (2 bytes)
  - Data (variable size)
- After aggregation, attribute is gzipped

Conclusions

- (KC) Profiling information is very useful, especially if JIT does not collect it itself
- (ANH) A lot of optimizations can be encoded in bytecode-with-register-allocation if you control the JIT
- Some care about security and code size
- Encoding results of other analyses in a general way has not yet been done (to my knowledge)