Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler

Kazuaki Ishizaki
IBM Research, Tokyo Research Laboratory

Goal of This Presentation

- Identify a set of optimizations that are cost-effective in the Just-In-Time (JIT) compiler across multiple platforms.
  - Cost: compilation time
  - Effectiveness: performance improvement

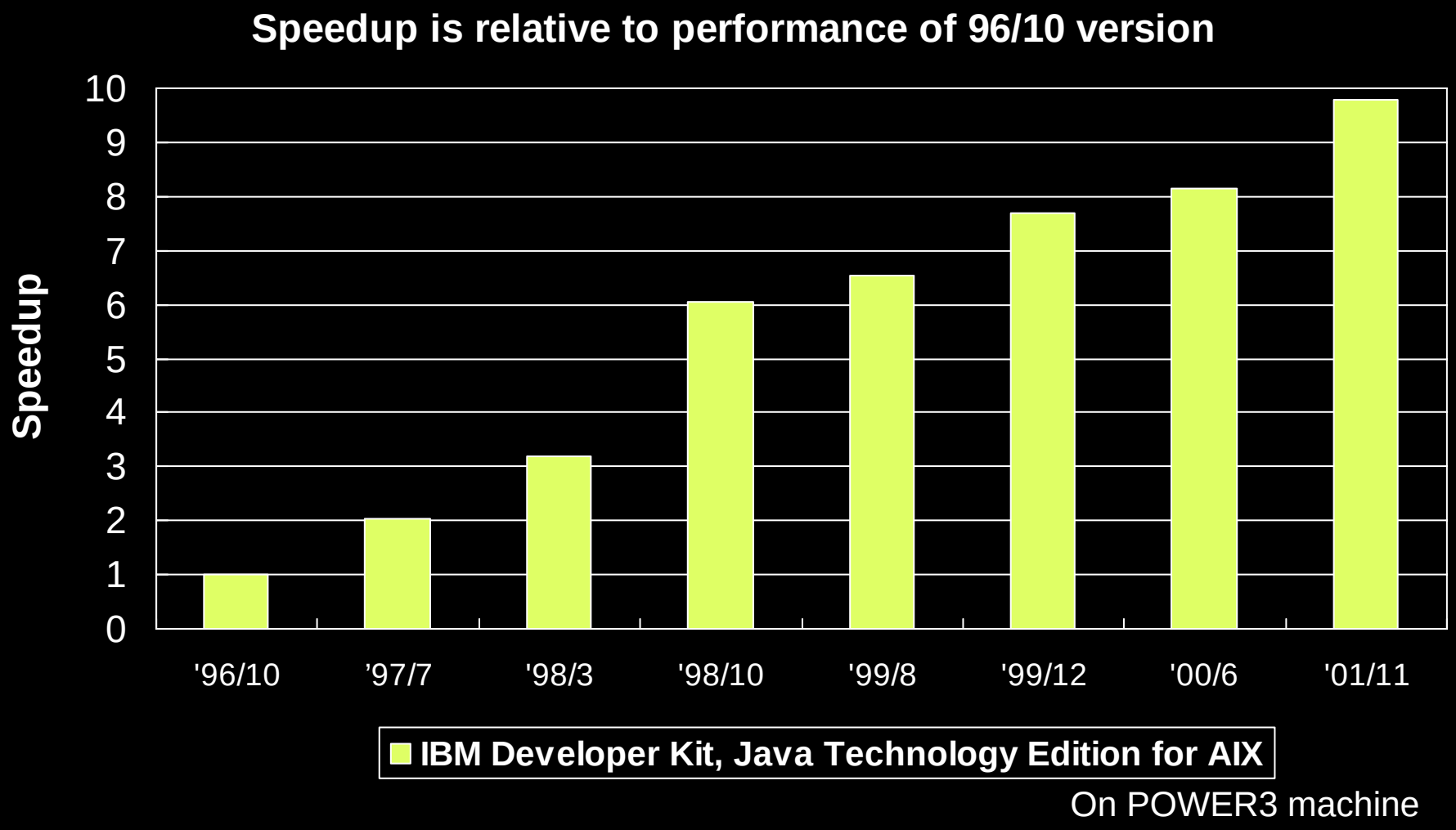
Contents

- Goal
- Overview of IBM Java JIT Compiler
- Individual Optimizations in the JIT Compiler
- Experimental Evaluations
  - Classify optimizations in terms of the balance between compilation time and performance improvement
- Summary

Summary of IBM Java JIT Compiler

- One of the industry-leading Java JIT compilers
- Perform a number of conventional and advanced optimizations for hot methods.
  - The interpreter executes other methods.
- Support a wide range of platforms
  - IA-32, Windows, Linux, and OS/2
  - IA-64, Windows and Linux
  - 32/64-bit PowerPC, AIX, Linux, and OS/400
  - 31/64-bit S/390, OS/390 and Linux

Performance improvement for SPECjvm98



Research outcome

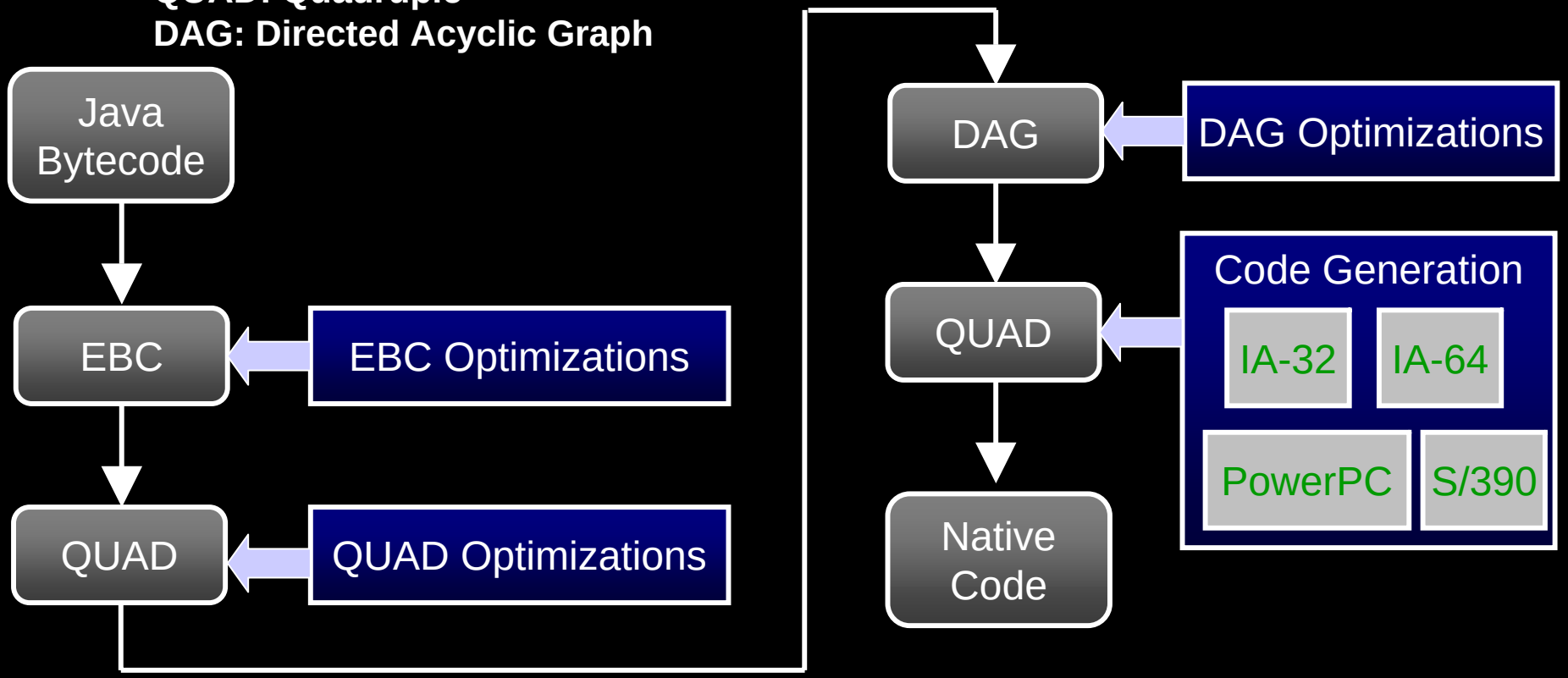
Please visit
http://www.research.ibm.com/trl/projects/jit/pub_int.htm

- JIT compiler
  - Method invocation optimization[OOPSLA00][JVM02]
  - Exception optimization[ASPLOS00][OOPSLA01][PACT02]
  - Profiling based optimization[JG00][PLDI03][PACT03]
  - Float optimization[JVM02][ICS02]
  - 64bit architecture optimization[PLDI02]
  - Register allocation[PLDI02]
  - Data prefetch[PLDI03]
  - Instruction Scheduling[CGO03]
  - Compiler overview[JG99][IBMSysJournal00][OOPSLA03]
- Runtime systems
  - Fast lock[OOPSLA99][OOPSLA02][ECOOP04]
  - Fast interpreter[ASPLOS02]

Flow of IBM JIT Compiler

- Use three types of intermediate representations

EBC: Extended Bytecode
QUAD: Quadruple
DAG: Directed Acyclic Graph



Three intermediate representations

	Features	Facilitated or intended Optimizations
Extended bytecode (EBC)	▪ Bytecode augmented with attribute information (type, resolusion status, …) ▪ Stack-based ▪ The most compact representation	Method inlining
Quadruple (QUAD)	▪ Tuple of opcode and zero or more operands ▪ Register-based ▪ Finer-grained semantics	Dataflow optimizations, including escape analysis and partial redundancy eliminations.
Directed acyclic graph (DAG)	▪ Data and exception dependences are represented	Loop optimizations and code scheduling.

An example of transformation

Java program

```
…
x = a[i];
…
```

Java bytecode

```
aload    4
iload    5
iaload
istore   6
```

Variable number	Symbolic name
4	a
5	i
6	x

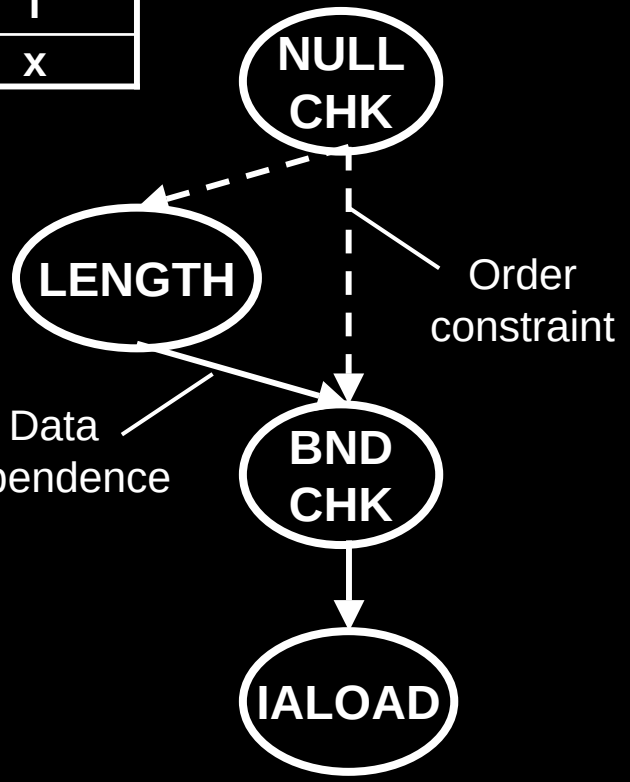
EBC

```
aload    4
iload    5
iaload        [null][bnd]
istore   6
```

QUAD

```
NULLCHK        LA4
LENGTH         LI7 = LA4
BNDCHK               LI5, LI7
IALOAD         LI6 = LA4,LI5*4+8
```

DAG



An example of code generation

QUAD

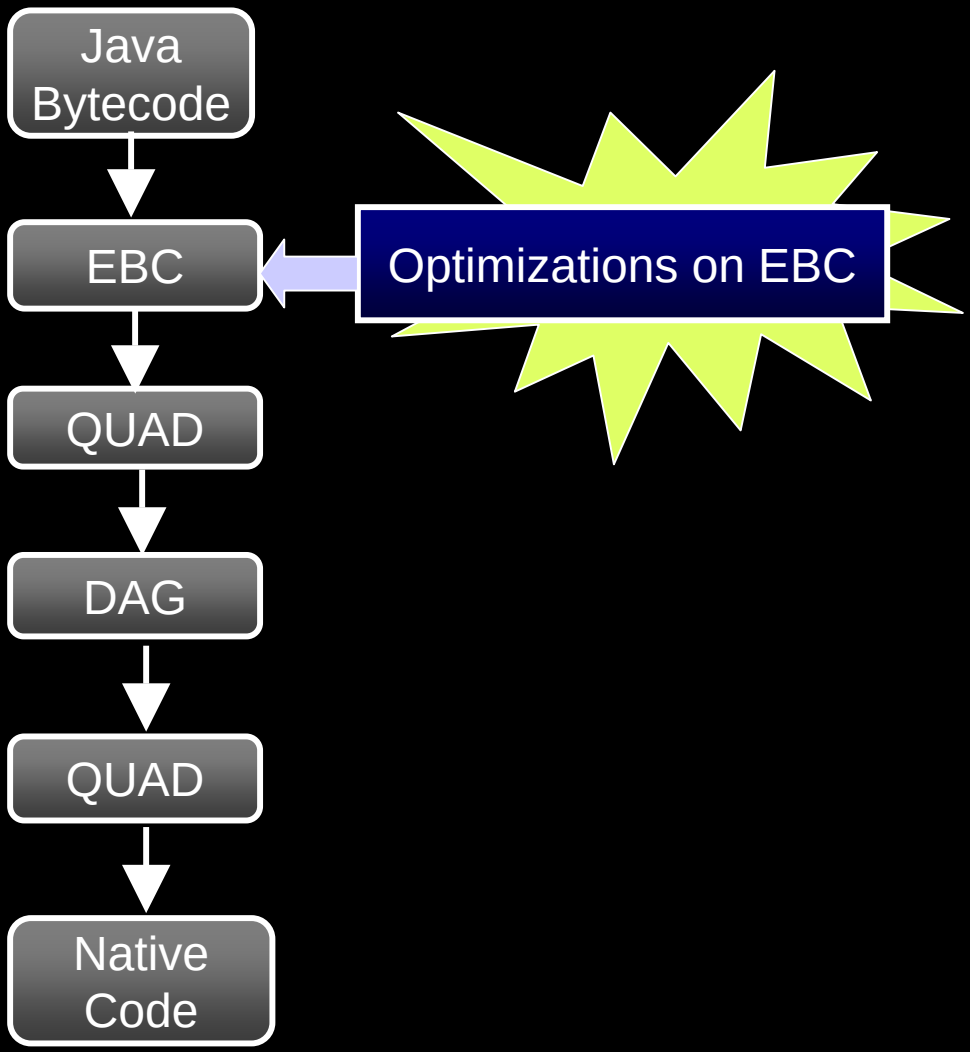
```
NULLCHK     LA4
LENGTH      LI7 = LA4
BNDCHK            LI5, LI7
IALOAD      LI6 = LA4,LI5*4+8
```

Variable number	Symbolic name
4	a
5	i
6	x

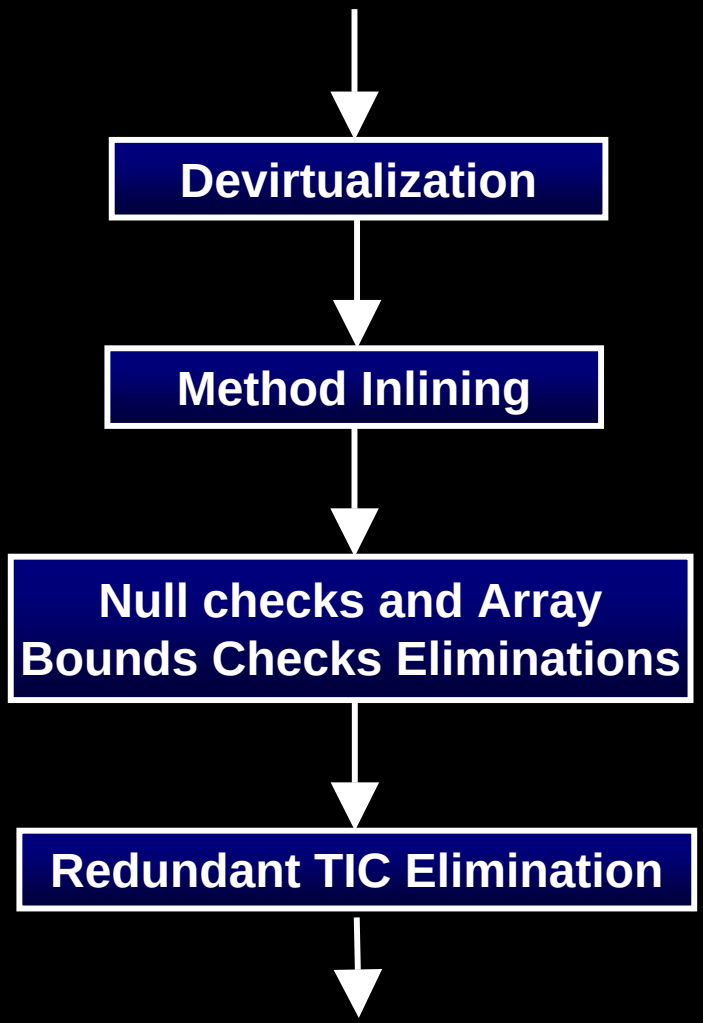
Native code (IA32)

```
...                                          : eax = LA4, edx = LI5
mov     ecx, [eax]                           : NULLCHK, LENGTH, ecx = LI7
cmp     edx, ecx                             : BNDCHK
jae     ThrowArrayIdxOutOfBndExcp            : BNDCHK
mov     ecx, [eax+edx*4+8]                   : IALOAD, ecx = LI6
```

Optimizations on EBC

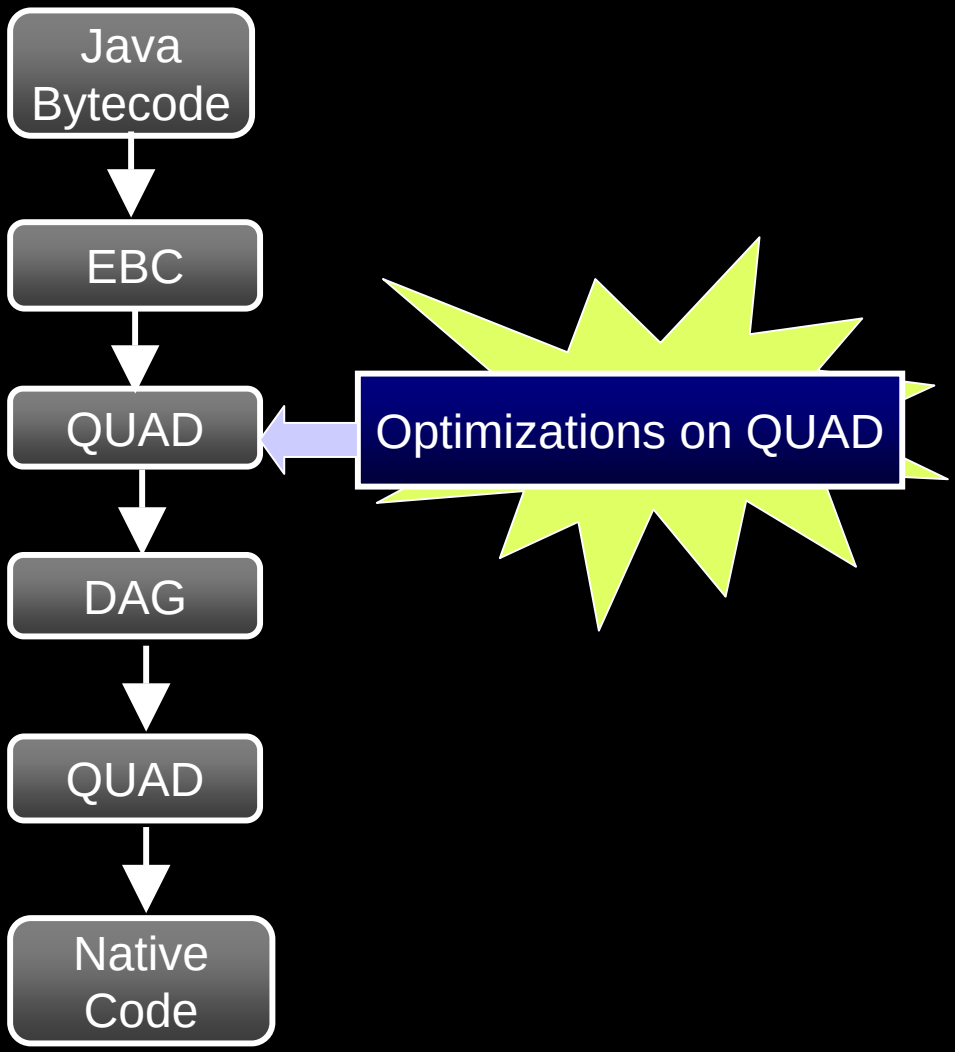


Optimizations on EBC

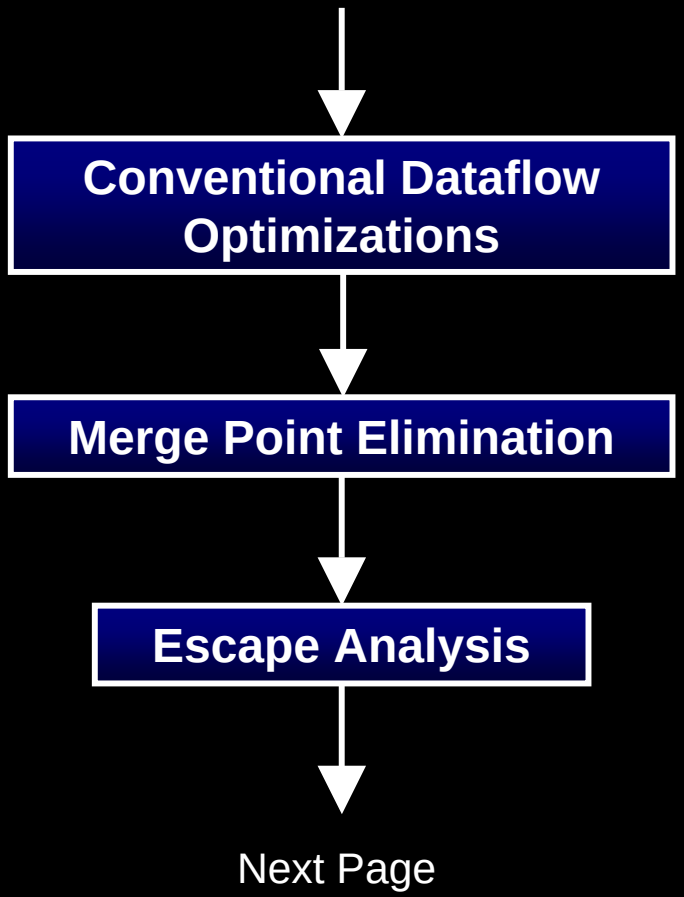


- Replace virtual calls with non-virtual calls if possible [Ishizaki2000oopsla].
- Use different budgets for small methods and non-small methods [Suganuma2002jvm].
- Use forward dataflow analysis.
- Use type flow analysis.
  - TIC: type inclusion check

Optimizations on QUAD

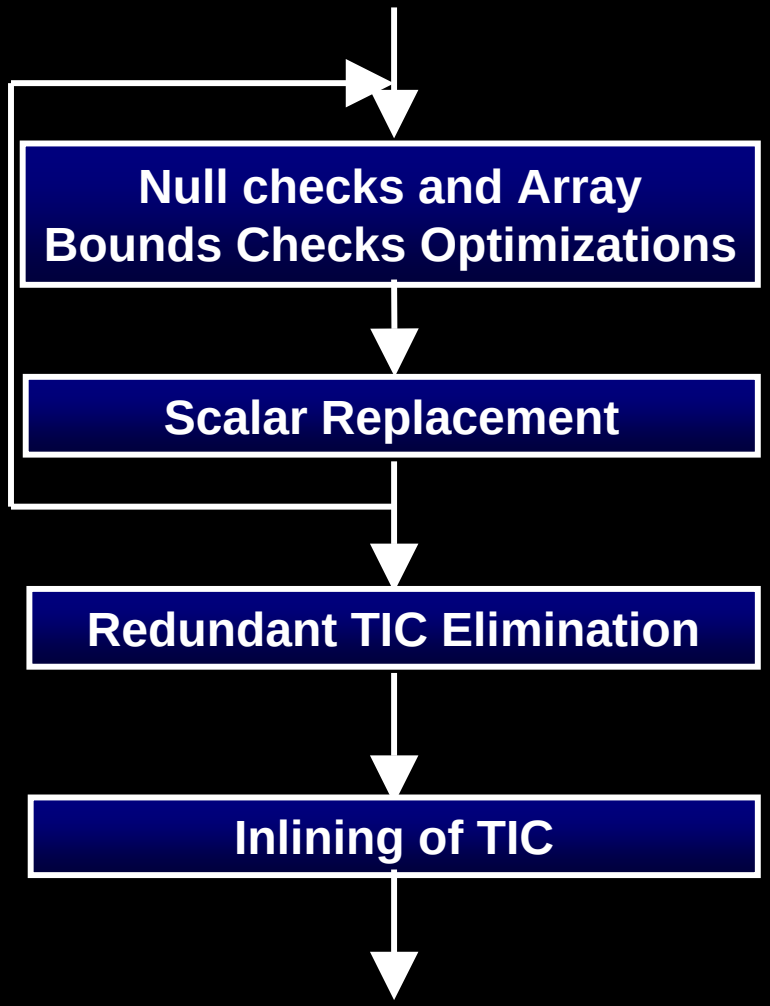


Optimizations on QUAD (1/2)



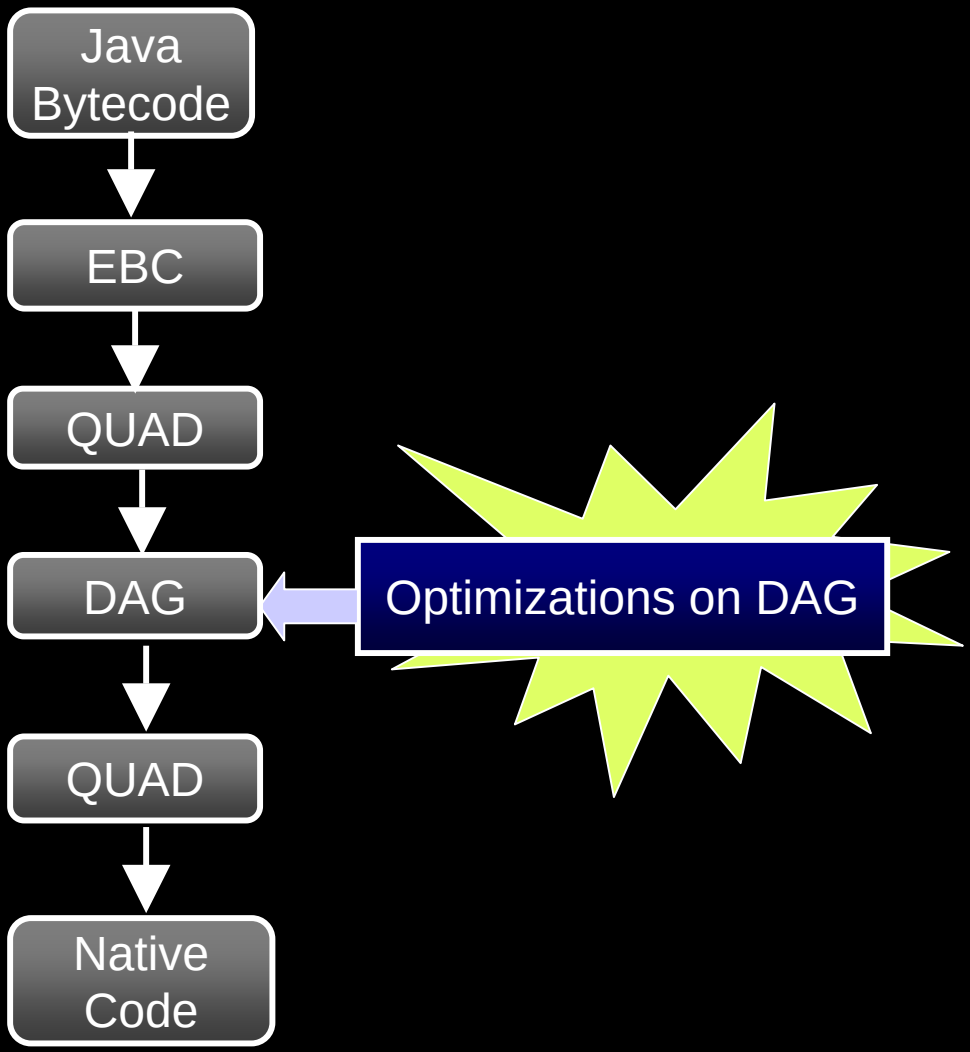
- Perform copy and constant propagation, and dead code elimination.
- Apply tail duplication.
- Examine whether an object escapes from a method or a thread [Whaley99oopsla].

Optimizations on QUAD (2/2)

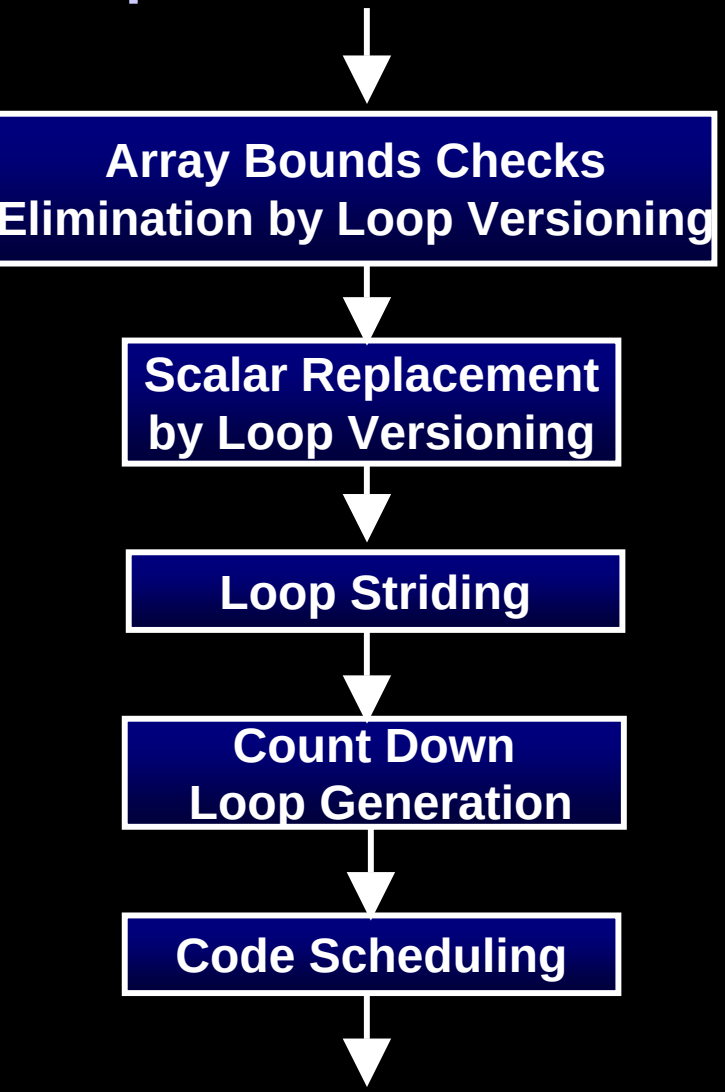


- Eliminate and move checks using partial redundancy elimination (PRE) [Kawahito2000asplos].
- Map instance and class variables to stack variables using PRE if possible.
- Use type flow analysis.
- Inline frequently executed paths of TICs [ishizaki99javagrande].

Optimizations on DAG

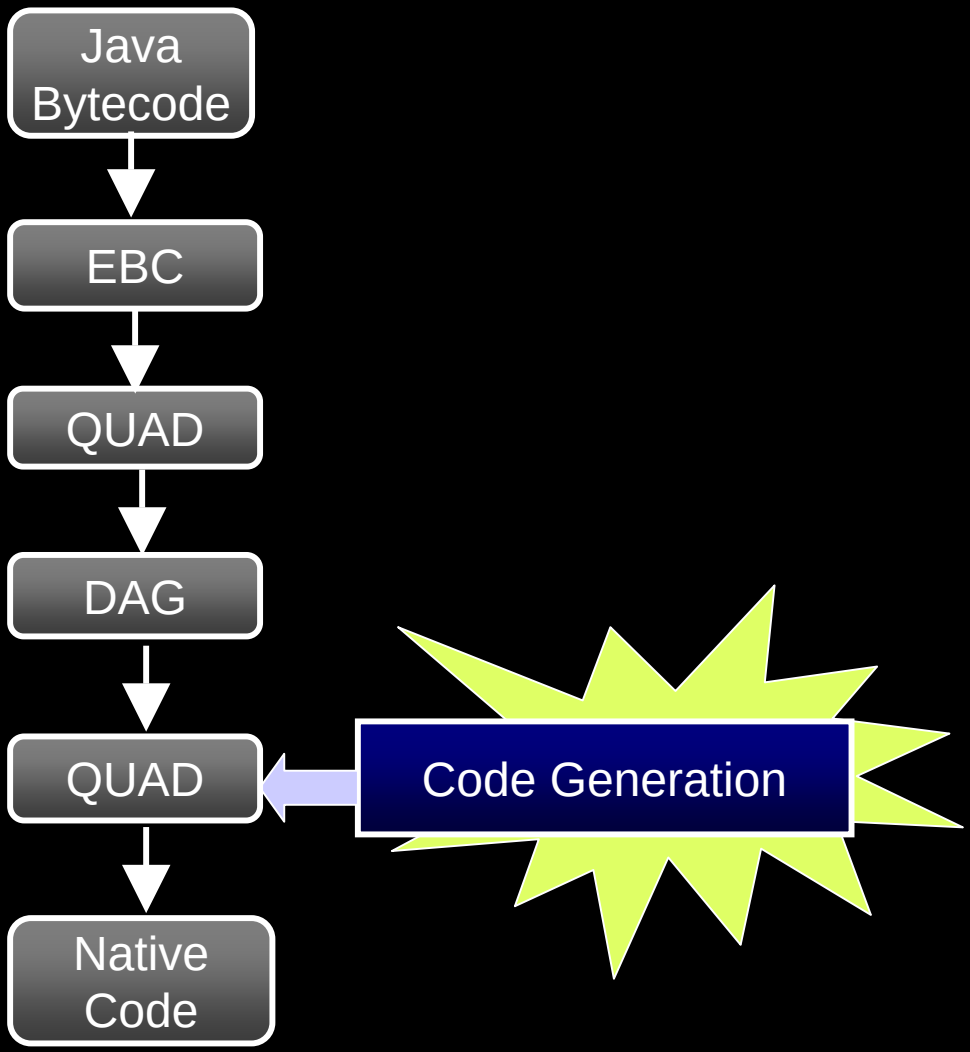


Optimizations on DAG

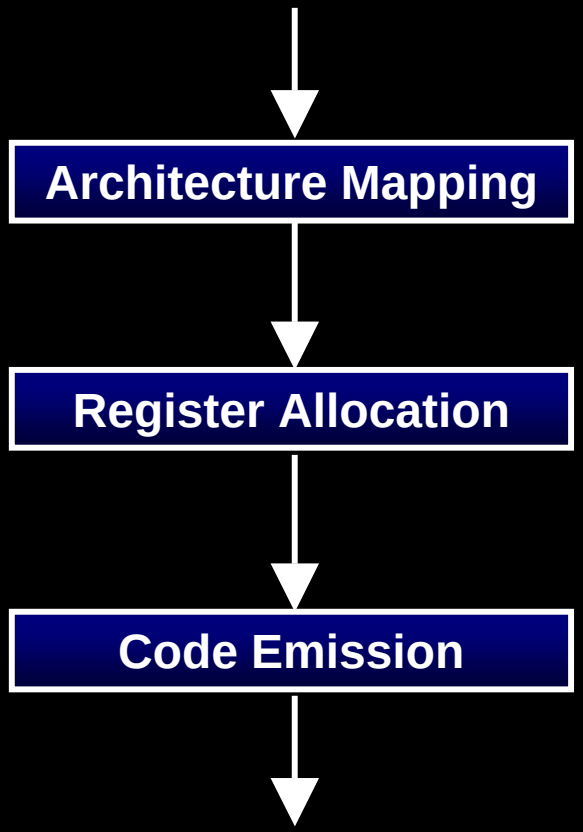


- Generate the original version loop, and the optimized version loop without array bounds checks [Suganuma2000ibmjournal].
- Generate the original version loop, and the optimized version loop without aliasing.
- Exploit instructions with incremental addressing mode (IA-64 and PowerPC).
- Exploit special loop count registers (IA-64 and PowerPC).
- Perform pre-pass code scheduling by a list scheduling.

Code Generation



Code Generation



- Impose architecture-specific limitations.
  - e.g. two-operands format for IA-32
    predicated code for IA-64
- Exploit architecture-specific features.
  - e.g. hyperblock for IA-64
- Assign physical registers.
- Generate machine instructions with post-pass first-fit code scheduling.

Experimental Environments

- Virtual Machine and JIT Compiler
  - IBM Developers Kit, Java Technology Edition, 1.4.0
  - Invoke JIT compiler for a method after the method is executed 1,000 times.
- Machines
  - IA-32

    2-Way 2.8GHz Xeon with 1GB memory, Windows 2000
  - IA-64

    2-Way 800MHz Itanium with 2GB memory, Windows .NET server
  - PowerPC (PPC)

    4-Way 1GHz POWER4 with 2GB memory, AIX 5.1L

Experimental Environments

- Benchmarks
  - SPECjvm98 (seven programs), size =100
  - SPECjbb2000 (one program), warehouse = 1

Experimental Evaluation

- Measure the effectiveness and the cost of each optimization *o* on multiple platforms (IA-32, IA-64, and PPC).
  - By disabling *o*.
  - *Effectiveness of o =*
    performance improvement (all enabled)
    – performance improvement (o disabled)
  - *Cost of o = compilation time (all enabled)*
    – compilation time (o disabled)

The Effectiveness of Optimizations

By performance improvement

Generally effective	Occasionally effective	Not effective
On all platforms, more than half of programs shows more than 4% performance improvement.	On some platform, some program shows more than 4% performance improvement.	No program shows more than 4% performance improvement.

The Cost of Optimizations

By compilation time

Small	On all platforms, increase compilation time by no more than 8%
Large	Increase compilation time by more than 8%

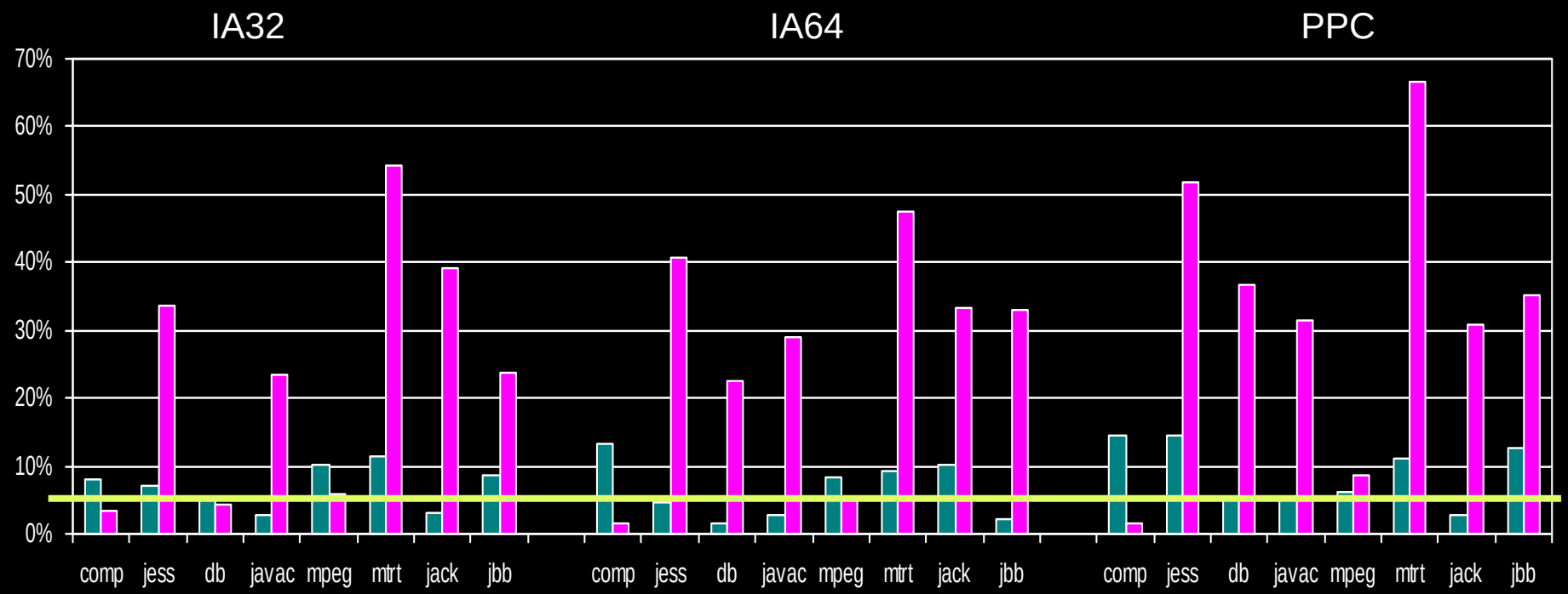
Which class does each optimization belong to ?

By performance improvement

By compilation time	Generally effective	Occasionally effective	Not effective
Small	Class A	Class B ???	Class C
Large	Class D	Class E	Class F

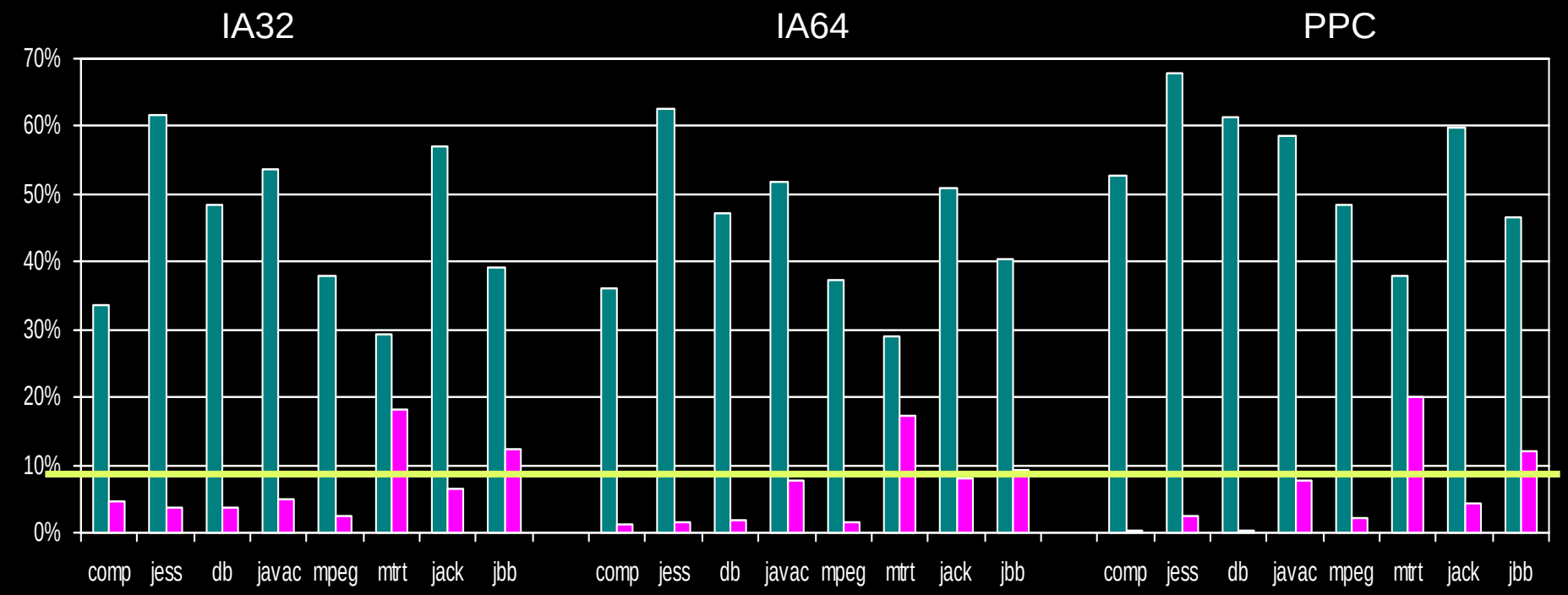
Optimizations: Method Inlining, Exception checks eliminations, Scalar replacement, Merge point Elimination, Escape analysis, DAG optimizations …

Classifying Method Inlining by the Effectiveness



	IA32	IA64	PPC	
Method inlining of non-small methods — Over 4%:	5 out of 8	4 out of 8	5 out of 8	➡ Occasionally effective
Method inlining of small methods — Over 4%:	5 out of 8	6 out of 8	7 out of 8	➡ Generally effective

Classifying Method Inlining by the Cost



Over 8%: Yes (avg. 41%) ➡ **Large**

Over 8%: No (avg. 6%) ➡ **Small**

Classifying optimizations - Results

By performance improvement

By compilation time	Generally effective	Occasionally effective	Not effective
Small	- Inlining of small methods - Exception checks elimination - Scalar replacement by PRE - Inlining of TICs	- Exception checks optimizations - Redundant TICs elimination - Merge points elimination	***ow risk, high return*** NONE
Large	NONE ***High risk, low return***	- Inlining of non-small methods - Escape analysis - DAG optimizations	- DAG optimizations

Small-Cost Optimizations

- Generally effective (Class A)
  - Inlining of small methods
  - Null checks and array bounds checks elimination (with only forward dataflow analysis)
  - Scalar replacement by PRE
  - Inlining of TICs
- Occasionally effective (Class B)
  - Null check and array bounds checks optimizations (with full analysis and PRE)
  - Redundant TICs elimination
  - Merge points elimination
- Not effective (Class C) - NONE

Large-Cost Optimizations

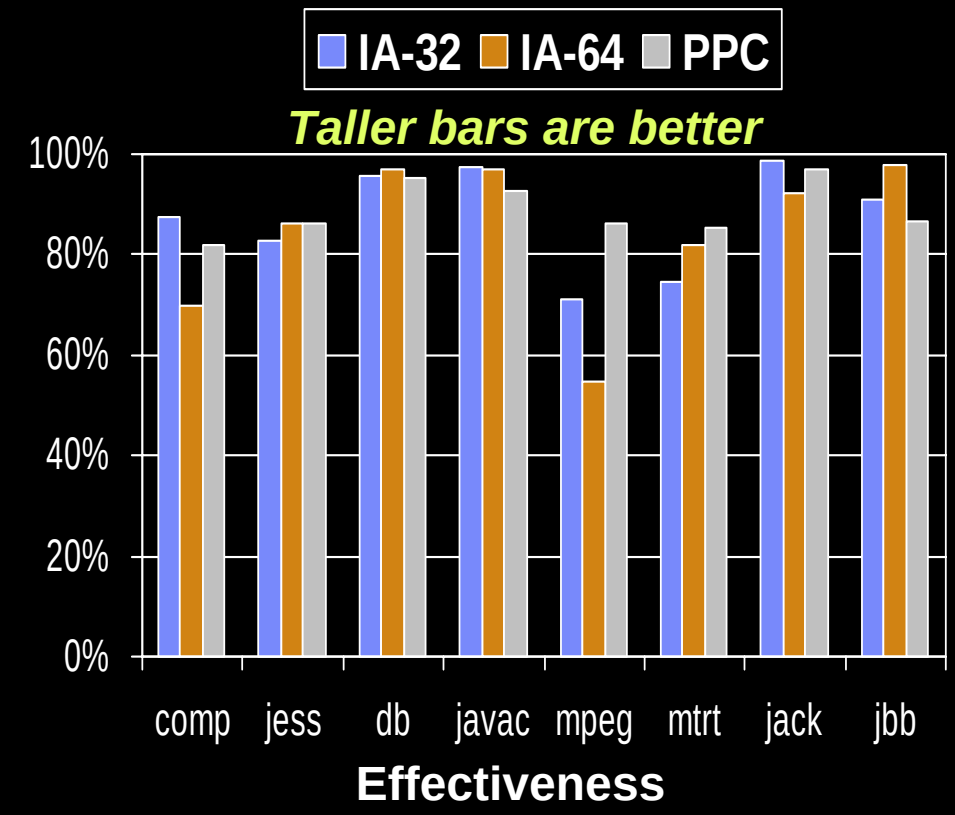
- Generally effective (Class D) - NONE
- Occasionally effective (Class E)
  - Inlining of non-small methods
  - Escape analysis
  - DAG Optimizations

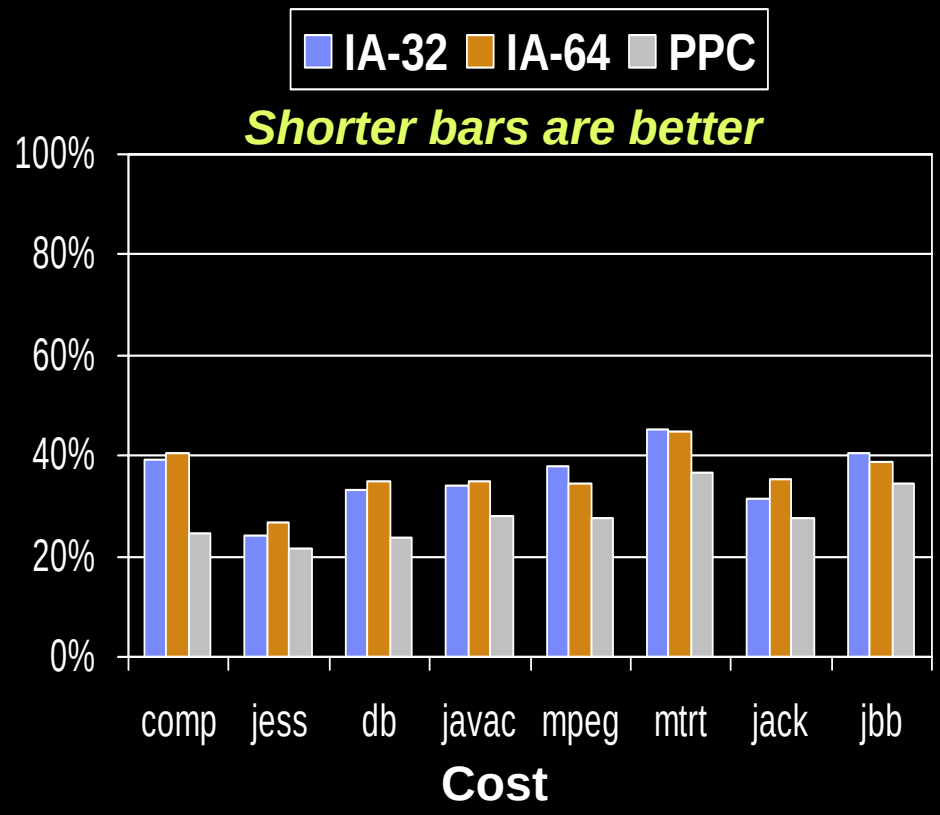
    Array bounds checks elimination by loop versioning and code scheduling
- Not effective (Class F)
  - DAG Optimizations

    Scalar replacement, loop striding, and count down loop generation

Results of Class A Optimizations

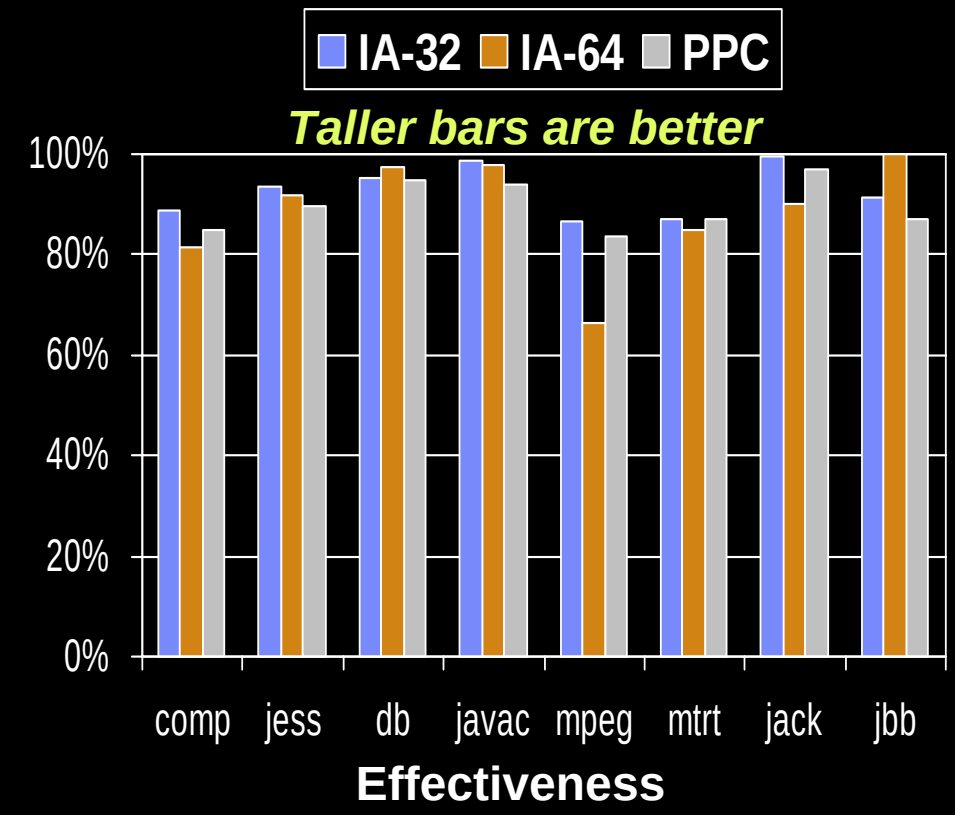
- Relative to all optimizations enabled,
  - Achieved 86% of the effectiveness
  - Spent 33% of the cost

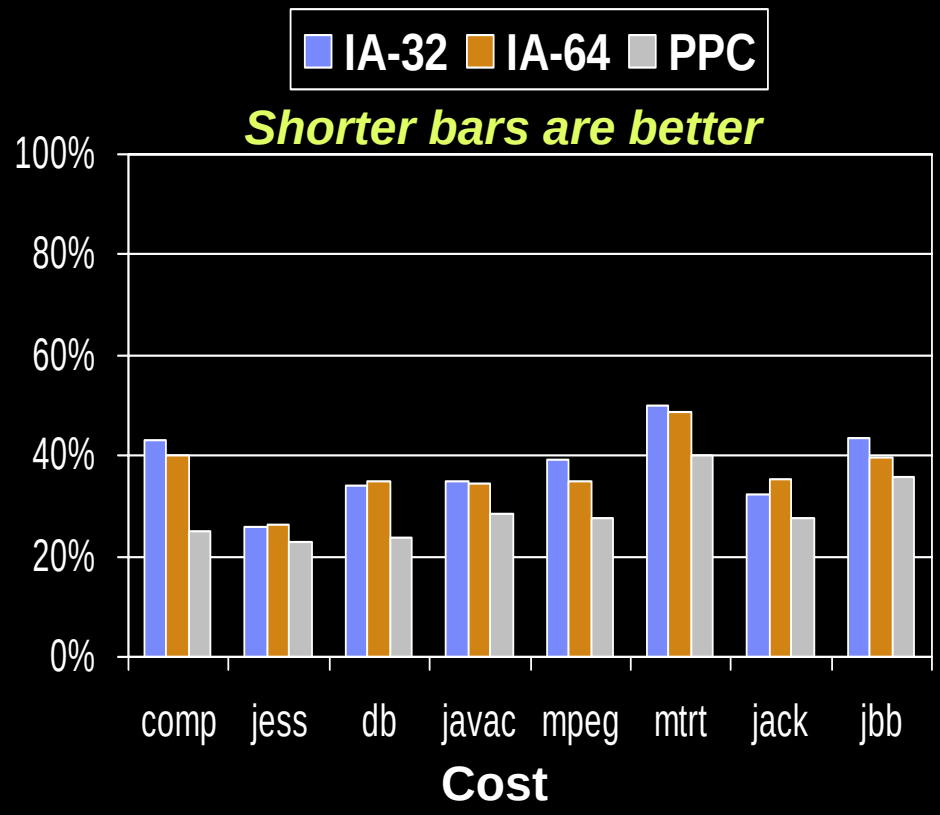




Results of Class A and B Optimizations

- Relative to all optimizations enabled,
  - Achieved 90% of the effectiveness
  - Spent 34% of the cost





Summary

- We identified a set of cost-effective optimizations (Class A+B) that achieved 90% of the effectiveness at 34% of the cost.
  - A Inlining of small methods
  - A Null checks and array bounds checks elimination (with only forward dataflow analysis)
  - A Scalar replacement by PRE
  - A Inlining of TICs
  - B Null checks and array bounds checks optimizations (with full analysis and PRE)
  - B Redundant TICs elimination
  - B Merge points elimination
- We will utilize the results to determine a set of optimizations for multi-level optimizations.

プロジェクトメンバー

- 中谷登志男
- 小松秀昭
- 小野寺民也
- 菅沼俊夫
- 河内谷清久仁
- 石崎一明
- 小笠原武史
- 川人基弘
- 安江俊明
- 竹内幹雄
- 緒方一則
- 古関聰
- 稲垣達氏

ありがとうございました。