Dynamic Compilation and Adaptive Optimization in Virtual Machines

Instructor: Michael Hind

Material contributed by:
Matthew Arnold, Stephen Fink, David Grove, and Michael Hind

Who am I?

- Helped build Jikes RVM (1998-2006)
  - GC Maps, live analysis, dominators, register allocation refactoring
  - Adaptive optimization system
  - Management, project promotion, education, etc.

- Work for IBM, home of 2 other Java VMs
  - IBM DK for Java, J9

- In previous lives, worked on
  - Automatic parallelization (PTran)
  - Ada implementation (Phd Thesis)
  - Interprocedural ptr analysis
  - Professor for 6 years

- Excited to share what I know
  - And learn what I don't!

Course Goals

- Understand the optimization technology used in production virtual machines
- Provide historical context of dynamic/adaptive optimization technology
- Debunk common misconceptions
- Suggest avenues of future research

Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

Course Outline

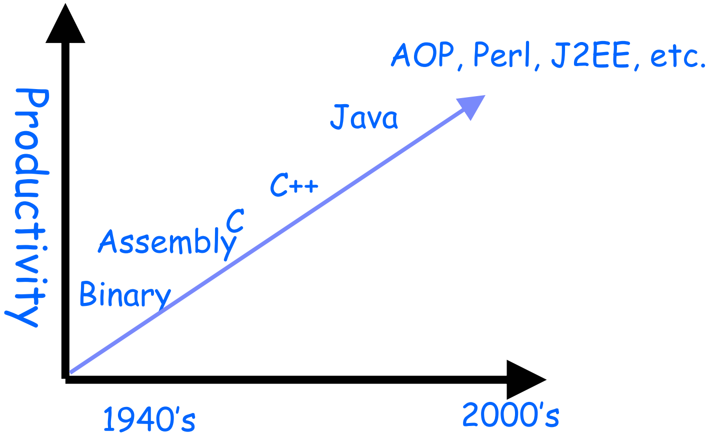
1. Background
   - Why software optimization matters
   - Myths, terminology, and historical context
   - How programs are executed
2. Engineering a JIT Compiler
   - What is a JIT compiler?
   - Case studies: Jikes RVM, IBM DK for Java, HotSpot
   - High level language-specific optimizations
   - VM/JIT interactions
3. Adaptive Optimization
   - Selective optimization
   - Design: profiling and recompilation
   - Case studies: Jikes RVM and IBM DK for Java
   - Understanding system behavior
   - Other issues
4. Feedback-Directed and Speculative Optimizations
   - Gathering profile information
   - Exploiting profile information in a JIT
     - Feedback-directed optimizations
     - Aggressive speculation and invalidation
   - Exploiting profile information in a VM
5. Summing Up and Looking Forward
   - Debunking myths
   - The three waves of adaptive optimization
   - Future directions

Course Outline - Summary

1. Background
    - Why software optimization matters
    - Myths, terminology, and historical context
    - How programs are executed
2. Engineering a JIT Compiler
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

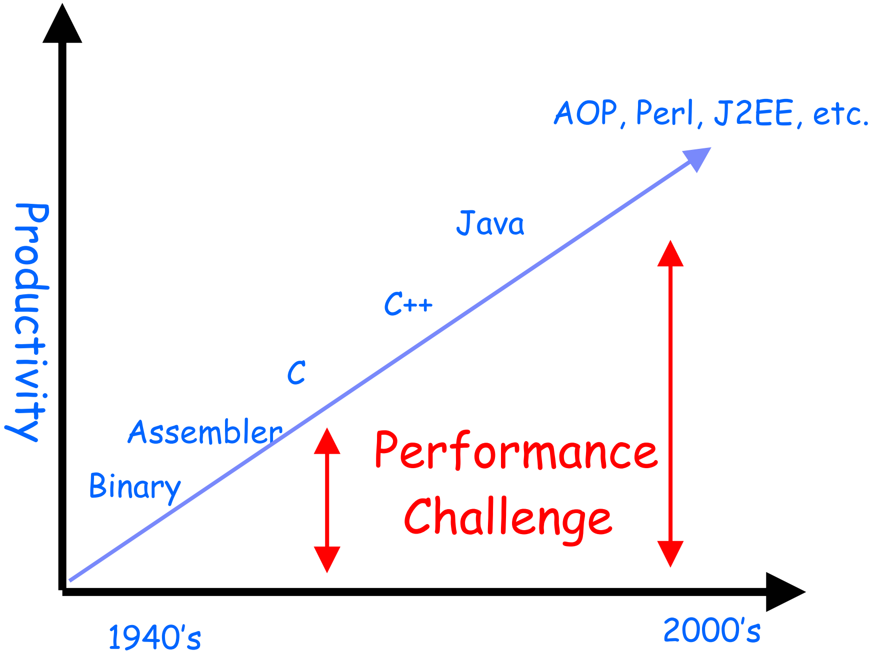
Developing Sophisticated Software

- Software development is difficult
- PL & SE innovations, such as
  - Dynamic memory allocation, object-oriented programming, strong typing, components, frameworks, design patterns, aspects, etc.
- Resulting in modern languages with many benefits
  - Better abstractions
  - Reduced programmer efforts
  - Better (static and dynamic) error detection
  - Significant reuse of libraries
- Have helped enable the creation of large, sophisticated applications



The Catch

- Implementing these features pose performance challenges
  - Dynamic memory allocation
    - Need pointer knowledge to avoid conservative dependences
  - Object-oriented programming
    - Need efficient virtual dispatch, overcome small methods, extra indirection
  - Automatic memory management
    - Need efficient allocation and garbage collection algorithms
  - Runtime bindings
    - Need to deal with unknown information
  - . . .



- Features require a rich runtime environment ➔ virtual machine

Type Safe, OO, VM-implemented Languages Are Mainstream

- Java is ubiquitous
  - eg. Hundreds of IBM products are written in Java
- “Very dynamic” languages are widespread and run on a VM
  - eg. Perl, Python, PHP, etc.
- These languages are not just for traditional applications
  - Virtual Machine implementation, eg. Jikes RVM
  - Operating Systems, eg. Singularity
  - Real-time and embedded systems, eg. Metronome-enabled systems
  - Massively parallel systems, eg. DARPA-supported efforts at IBM, Sun, and Cray
- Virtualization is everywhere
  - browsers, databases, O/S, binary translators, VMMs, in hardware, etc.

Have We Answered the Performance Challenges?

- So far, so good …
  - Today's typical application on today's hardware runs as fast as 1970s typical application on 1970s typical hardware
  - Features expand to consume available resources…
  - eg. Current IDEs perform compilation on every save

- Where has the performance come from?
  1. Processor technology, clock rates (X%)
  2. Architecture design (Y%)
  3. Software implementation (Z%)

  X + Y + Z = 100%

- HW assignment: determine X, Y, and Z

Future Trends - Software

- Software development is still difficult
  - PL/SE innovation will continue to occur
  - Trend toward more late binding, resulting in dynamic requirements
  - Will pose further performance challenges

- Real software is now built by piecing components together
  - Components themselves are becoming more complex, general purpose
  - Software built with them is more complex
    - Application server (J2EE Websphere, etc), application framework, standard libraries, non-standard libraries (XML, etc), application
  - Performance is often terrible
    - J2EE benchmark creates 10 business objects (w/ 6 fields) from a SOAP message [Mitchell et al., ECOOP'06]
      - > 10,000 calls
      - > 1,400 objects created
  - Traditional compiler optimization wouldn't help much
    - Optimization at a higher semantic level could be highly profitable

Future Trends - Hardware

- Processor speed advances not as great as in the past (x << X?)
- Computer architects providing multicore machines
  - Will require software to utilize these resources
  - Not clear if it will contribute more than in the past (y ? Y)
- Thus, one of the following will happen
  - Overall performance will decline
  - Increase in software sophistication will slow
  - Software implementation will pick up the slack (z > Z)

Future Trends – Hardware

- Processor speed advances not as great as in the past (x << X?)
- Computer architects providing multicore machines
  - Will require software to utilize these resources
  - Not clear if it will contribute more than in the past (y ? Y)
- Thus, one of the following will happen
  - Overall performance will decline
  - Software complexity growth will slow
  - **Software implementation will pick up the slack (z > Z)**

Course Outline

1. Background
    - Why software optimization matters
    - **Myths, terminology, and historical context**
    - How programs are executed
2. Engineering a JIT Compiler
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

Well-Known "Facts"

1. Because they execute at runtime, dynamic compilers must be blazingly fast
2. Dynamic class loading is a fundamental roadblock to cross-method optimization
3. Sophisticated profiling is too expensive to perform online
4. A static compiler will always produce better code than a dynamic compiler
5. Infrastructure requirements stifle innovation in this field
6. Production VMs avoid complex optimizations, favoring stability over performance

Terminology

Virtual Machine (for this talk): a software execution engine for a program written in a machine-independent language

- Ex., Java bytecodes, CLI, Pascal p-code, Smalltalk v-code

VM != JIT

Adaptive Optimization Hall of Fame

- 1958-1962
- 1974
- 1980-1984
- 1986-1994
- 1995-present

Adaptive Optimization Hall of Fame

- 1958-1962: LISP
- 1974: **Adaptive Fortran**
- 1980-1984: **ParcPlace Smalltalk**
- 1986-1994: **Self**
- 1995-present: **Java**

Quick History of VMs

- LISP Interpreters [McCarthy'78]
  - First widely used VM
  - Pioneered VM services
    - memory management
    - *Eval* -> dynamic loading
- Adaptive Fortran [Hansen'74]
  - First in-depth exploration of adaptive optimization
  - Selective optimization, models, multiple optimization levels, online profiling and control systems

Quick History of VMs

- ParcPlace Smalltalk [Deutsch&Schiffman'84]
  - First modern VM
  - Introduced full-fledge JIT compiler, inline caches, native code caches
  - Demonstrated software-only VMs were viable

- Self [Chambers&Ungar'91, Hölzle&Ungar'94]
  - Developed many advanced VM techniques
  - Introduced polymorphic inline caches, on-stack replacement, dynamic de-optimization, advanced selective optimization, type prediction and splitting, profile-directed inlining integrated with adaptive recompilation

- Java/JVM [Gosling et al. '96]
  - First VM with mainstream market penetration
  - Java vendors embraced and improved Smalltalk and Self technology
  - Encouraged VM adoption by others -> CLR

Featured VMs in this Talk

- Self ['86-'94]
  - Self is a pure OO language
  - Supports an interactive development environment
  - Much of the technology was transferred to Sun's HotSpot JVM
- IBM DK for Java ['95-'06]
  - Port of Sun Classic JVM + JIT + GC and synch enhancements
  - Compliant JVM
  - World class performance
- Jikes RVM (Jalapeño) ['97-]
  - VM for Java, written in (mostly) Java
  - Independently developed VM + GNU Classpath libs
  - Open source, popular with researchers, not a compliant JVM

Course Outline

1. Background
   - Why software optimization matters
   - Myths, terminology, and historical context
   - **How programs are executed**
2. Engineering a JIT Compiler
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

How are Programs Executed?

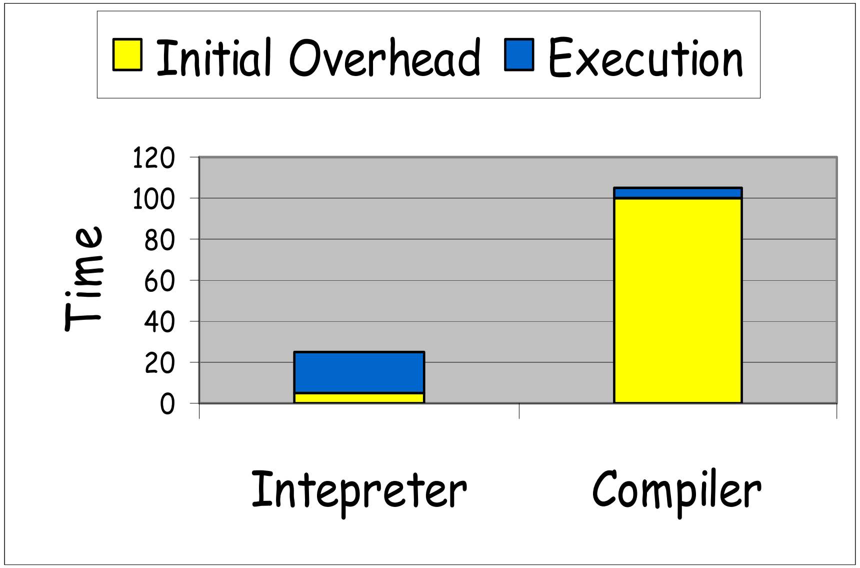
1. Interpretation
   - Low startup overhead, but much slower than native code execution
     - Popular approach for high-level languages
       - Ex., APL, SNOBOL, BCPL, Perl, Python, MATLAB
     - Useful for memory-challenged environments
2. Classic just-in-time compilation
   - Compile each method to native code on first invocation
     - Ex., ParcPlace Smalltalk-80, Self-91
     - Initial high (time & space) overhead for each compilation
     - Precludes use of sophisticated optimizations (eg. SSA, etc.)

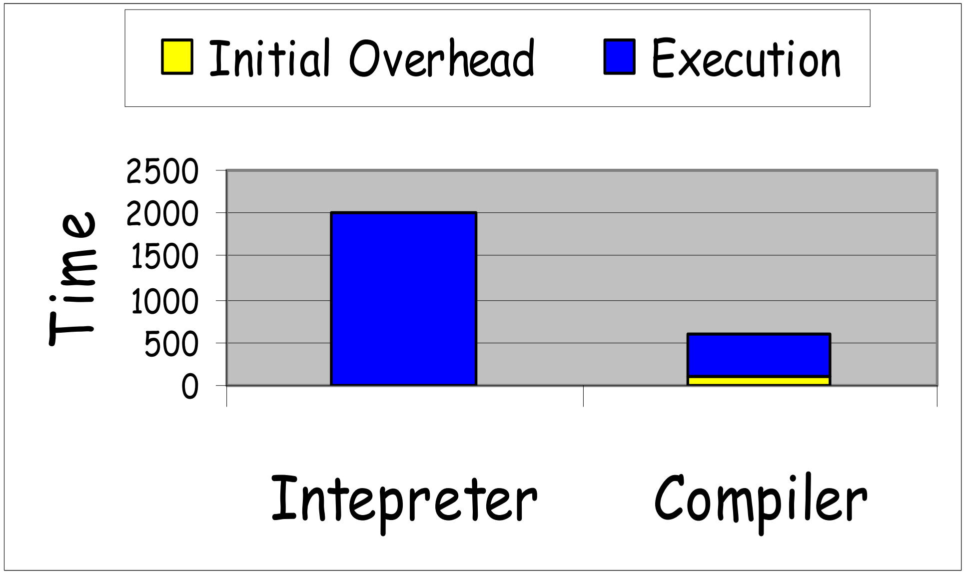
Responsible for many of today's misconceptions

Interpretation vs. (Dynamic) Compilation

Example: 500 methods

Assume: Compiler gives 4x speedup, but has 20x overhead





Short running: Interpreter is best

Long running: compilation is best

Selective Optimization

- Hypothesis: most execution is spent in a small pct. of methods
  - 90/10 (or 80/20) rule
- Idea: use two execution strategies
  1. **Unoptimized**: interpreter or non-optimizing compiler
  2. **Optimized**: Full-fledged optimizing compiler
- Strategy
  - Use unoptimized execution initially for all methods
  - Profile application to find “hot” subset of methods
    - Optimize this subset
    - Often many times

Course Outline

1. Background
2. **Engineering a JIT Compiler**
   - **What is a JIT compiler?**
   - Case studies: Jikes RVM, IBM DK for Java, HotSpot
   - High level language-specific optimizations
   - VM/JIT interactions
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

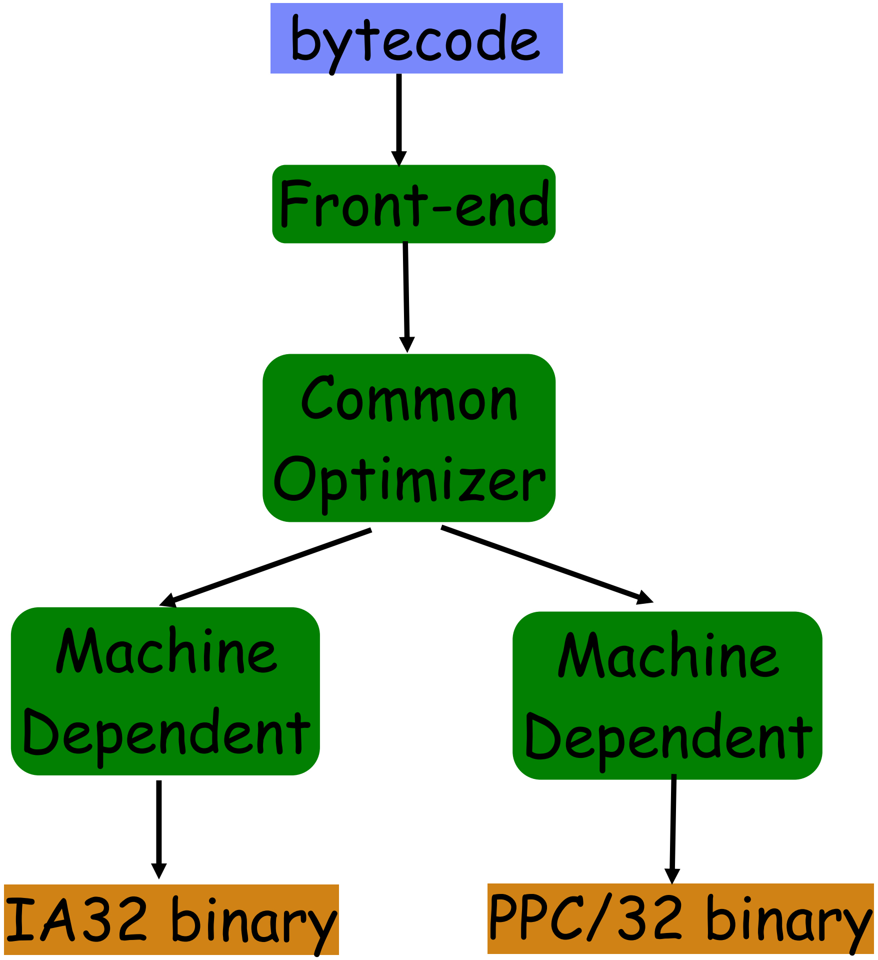
What is a JIT Compiler?

- Code generation component of a virtual machine
- Compiles bytecodes to in-memory binary machine code
  - Simpler front-end and back-end than traditional compiler
    - Not responsible for source-language error reporting
    - Doesn't have to generate object files or relocatable code
- Compilation is interspersed with program execution
  - Compilation time and space consumption are very important
- Compile program incrementally; unit of compilation is a method
  - JIT may never see the entire program
  - Must modify traditional notions of IPA (Interprocedural Analysis)

Design Requirements

- High performance (of executing application)
  - Generate "reasonable" code at "reasonable" compile time costs
  - Selective optimization enables multiple design points
- Deployed on production servers ➔ RAS
  - Reliability, Availability, Serviceability
  - Facilities for logging and replaying compilation activity
- Tension between high performance and RAS requirements
  - Especially in the presence of (sampling-based) feedback-directed opts
  - So far, a bias to performance at the expense of RAS, but that is changing as VM technology matures
    - Ogato et al., OOPSLA'06 discuss this issue

Structure of a JIT Compiler


bytecode

Front-end

Common Optimizer

Machine Dependent

Machine Dependent

IA32 binary

PPC/32 binary

Course Outline - Summary

1. Background
2. Engineering a JIT Compiler
   - What is a JIT compiler?
   - **Case studies: Jikes RVM, IBM DK for Java, HotSpot**
   - High level language-specific optimizations
   - VM/JIT interactions
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

Case Study 1: Jikes RVM [Fink et al., OOPSLA'02 tutorial]

- Java bytecodes ➔ IA32, PPC/32
- 3 levels of Intermediate Representation (IR)
  - Register-based; CFG of extended basic blocks
  - HIR: operators similar to Java bytecode
  - LIR: expands complex operators, exposes runtime system implementation details (object model, memory management)
  - MIR: target-specific, very close to target instruction set
- Multiple optimization levels
  - Suite of classical optimizations and some Java-specific optimizations
  - Optimizer preserves and exploits Java static types all the way through MIR
  - Many optimizations are guided by profile-derived branch probabilities

Jikes RVM Opt Level 0

- On-the-fly (bytecode → IR)
  - constant, type and non-null propagation, constant folding, branch optimizations, field analysis, unreachable code elimination
- BURS-based instruction selection
- Linear scan register allocation

- Inline trivial methods (methods smaller than a calling sequence)
- Local redundancy elimination (CSE, loads, exception checks)
- Local copy and constant propagation; constant folding
- Simple control flow optimizations
  - Static splitting, tail recursion elimination, peephole branch opts
- Simple code reordering
- Scalar replacement of aggregates & short arrays
- One pass of global, flow-insensitive copy and constant propagation and dead assignment elimination

Jikes RVM Opt Level 1

- Much more aggressive inlining
  - Larger space thresholds, profile-directed
  - Speculative CHA (recover via preexistence and OSR)
- Runs multiple passes of many level 0 optimizations
- More sophisticated code reordering algorithm [Pettis&Hansen]

- Over time many optimizations shifted from level 1 to level 0
- Aggressive inlining is currently the primary difference between level 0 and level 1

Jikes RVM Opt Level 2

- Loop normalization, peeling & unrolling
- Scalar SSA
  - Constant & type propagation
  - Global value numbers
  - Global CSE
  - Redundant conditional branch elimination
- Heap Array SSA
  - Load/store elimination
  - Global code placement (PRE/LICM)

Case Study 2: IBM DK [Ishizaki et al. '03]

- Java bytecodes ➔ IA32, IA64, PPC/32, PPC/64, S/390
- 3 Intermediate representations
  - Extended bytecodes (compact, but can't express all transforms)
  - Quadruples (register-based IR)
  - DAG (quadruples + explicit representation of all dependencies)
- Multiple optimization levels
- Many optimizations use profile information

Optimizations on Extended Bytecodes

- Java bytecodes + type information
  - Compact representation
  - Can't express some transformations
- Flow-sensitive type inference (devirtualization)
- Method inlining, includes guarded inlining based on CHA
- Nullcheck and array bounds check elimination
- Flow-sensitive type inference (checkcast/instanceof)

Optimizations on Quadruples

- Quadruples
  - Register-based; CFG of extended basic blocks
  - Close to native instruction set; some pseudo-operators (e.g. new)

- Copy and constant propagation, dead code elimination
- Frequency-directed splitting
- Escape analysis & scalar replacement
- Exception check optimization (partial-PRE)
- Type inference (instanceof/checkcast)

Optimizations on DAG of QUADs

- DAG: augment QUADs with explicit dependency edges

- SSA-form: loop versioning, induction variable optimizations
- Pre-pass instruction scheduling
- Instruction selection
- Sign extension elimination
- Code reordering (move infrequent blocks to end)
- Register allocation
  - Special-purpose for IA32
  - Linear scan other platforms
  - Considering graph coloring
- Post-pass instruction scheduling

Effectiveness of Optimizations in IBM DK [Ishizaki, et al. OOPSLA'03]

- Generally effective and cheap
  - Method inlining for tiny methods
  - Exception check elimination via forward dataflow
  - Scalar replacement via forward dataflow
- Sometimes effective and cheap
  - Exception check elimination via PRE
  - Elimination of redundant instanceof/checkcast
  - Splitting
- Occasionally effective, but expensive
  - Method inlining of larger methods via static heuristics
  - Scalar replacement via escape analysis
  - All of their DAG optimizations

Case Study 3: HotSpot Server JIT [Paleczny et al. '01]

- HotSpot Server compiler
  - Client compiler is simpler; small set of opts but faster compile time
- Java bytecodes ➔ SPARC, IA32
- Extensive use of On Stack Replacement
  - Supports a variety of speculative optimizations (more later)
  - Integral part of JIT's design
- Of the 3 systems, the most like an advanced static optimizer
  - SSA-form and heavy optimization
  - Design assumes selective optimization ("HotSpot")

HotSpot Server JIT

- Virtually all optimizations done on SSA-based sea-of-nodes
  - Global value numbering, sparse conditional constant propagation,
  - Fast/Slow path separation
  - Instruction selection
  - Global code motion [Click '95]
- Graph coloring register allocation with live range splitting
  - Approx 50% of compile time (but much more than just allocation)
  - Out-of-SSA transformation, GC maps, OSR support, etc.

Course Outline

1. Background
2. Engineering a JIT Compiler
   - What is a JIT compiler?
   - Case studies: Jikes RVM, IBM DK for Java, HotSpot
   - **High level language-specific optimizations**
   - VM/JIT interactions
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

High level language-specific optimizations

- Not a consequence of JIT compilation, but of source language
- Effective optimization of object-oriented language features is essential for high performance
- Optimizations
  - Type analysis: virtual function calls and typechecks
  - Escape analysis, scalar replacement, etc.
  - Support for precise exceptions

Optimizing Virtual Function Calls

- Effective inlining is the most important optimization in a JIT
  - Many small methods
  - Many virtual function calls (target not directly evident)
- Iterative Type Analysis [Chambers&Ungar'90]
  - Compute for every variable a conservative approximation of the runtime types (concrete types) of values stored in that variable
  - Gains information from new, checkcast, virtual call, …
  - Enables devirtualization (and then inlining)
  - Also can be used to eliminate redundant checkcast/instanceof
- Type analysis is useful, but often not sufficient

Speculatively Optimizing Virtual Function Calls

- Class Hierarchy Analysis [Dean et al. '95]
  - constrained by potential for dynamic class loading
  - guard with class/method test or code patch
  - avoid guards with preexistence or OSR
- Profile-guided
  - guard with class/method test
- More details later…

Optimization of Heap Allocated Objects

- "Good" OO programming ➔ heavy use of heap allocated objects
- Optimizations
  - Reduce direct cost of allocating objects
    - Inline allocation sequence, thread-local allocation pools
    - Stack allocation & scalar replacement of non-escaping objects
  - Support advanced GC algorithms (write barriers for generational)
  - Deeper analysis of load/stores to the heap
    - Eliminate redundant load/stores
    - Extend other analyses to cope with dataflow through instance variables

Scalar Replacement

- Completely replace all references to an object
- Enabled by escape analysis and/or dataflow

```
class A {
    int x;
    int y;
}
void foo() {
    A a = new A();
    a.x = 1;
    a.y = a.x + 2;
    System.out.println(a.y);
}
```

```
void foo() {
    int t1 = 1;
    int t2 = t1 + 2;
    System.out.println(t2);
}
```

Redundant Load Elimination

Original Program

```
p  := new Z
q  := new Z
r  := p
 . . .
p.x  := ...

q.x := ...
 ...   := r.x
```

Transformed Program

```
p  := new Z
q  := new Z
r  := p
 . . .
T1  := ...
p.x  := T1
q.x := ...
 ...   := T1
```

Optimizing with Precise Exceptions

- Language semantics require precise exception handling
  - Constrains optimizations by limiting legal reorderings of operations and may extend the lifetime of variables
  - Optimizations must be taught to respect these constraints
    - Principled: IR represents all constraints of exception model
    - Kludge: Special logic in every impacted optimization
    - Reality: combination of the two approaches
- Optimizations to reduce performance impact
  - Eliminate redundant exception checks
  - Hoist invariant checks; PRE of checks
  - Loop peeling and loop versioning to create fast loops for the expected case

Course Outline

1. Background
2. Engineering a JIT Compiler
    - What is a JIT compiler?
    - Case studies: Jikes RVM, IBM DK for Java, HotSpot
    - High level language-specific optimizations
    - **VM/JIT interactions**
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

JIT/VM Interactions

- Runtime services often require JIT support
  - Memory management
  - Exception delivery and symbolic debugging
- JITed code requires extensive runtime support
  - Runtime services such as type checking, allocation
  - Common to use hardware traps & signal handlers
  - Helper routines for uncommon cases (dynamic linking)
- Collaboration enables optimization opportunities
  - Inline common case of allocation, type checks, etc.
  - Co-design of VM & JIT essential for high performance

JIT Support for Memory Management

- GC Maps
  - Required for type-accurate GC to identify roots for collection
  - Generated by JIT for every program point where a GC may occur
  - Encodes which physical registers and stack locations hold objects
  - Can constrain optimizations (derived pointers)
- Write barriers for generational collection
  - Requires JIT cooperation (barriers inserted in generated code)
  - Common case of barriers is usually inlined
  - Variety of barrier implementations with different trade-offs
- Cooperative scheduling
  - In many VMs, all mutator threads must be stopped at GC points.
  - One solution requires JITs to inject GC yieldpoints at regular intervals in the generated code

JIT Support for Other Runtime Services

- Exception tables
  - Encode try/catch structure in terms of generated machine code.
  - Typical implementation in JVM consists of compact meta-data generated by the JIT and used when an exception occurs
    - no runtime cost when there is no exception
- Mapping from machine code to original bytecodes
  - Primary usage is for source level debugging, but if the mapping exists it can be used to support a variety of other runtime services
  - One complication is the encoding of inlining structure to present view of virtual call stack

Runtime Support for JIT Generated Code

- Memory allocation
  - Occurs frequently, therefore JIT usually inlines common case
  - Details of GC implementation often "leak" into the JIT making GC harder to maintain and change (some exceptions: Jikes RVM; LIL [Glew et al. VM'04])
- Null pointer checks; array bounds check
  - Implemented via SIGSEGV and/or trap instructions
  - Runtime installs signal handlers to handle traps and create/throw appropriate language level exception
- JIT generated code relies on extensive set of runtime helper routines
  - "Outline" infrequent operations and uncommon cases of frequent operations
  - Very common place for JIT details to "leak" into the runtime system and vice versa.
  - Often use specialized calling conventions for either fast invocation or reduced code space

Advantages of JIT/VM Interdependency

- Co-design of JIT/VM can have large performance implications
- VM data structures optimized to enable JIT to generate effective inline code sequences for common cases.
- Example: support for dynamic type checking in JVMs
  - Jikes RVM [Alpern et al.'01] and HotSpot [Click&Rose'02]
  - Similar ideas, HotSpot extends and improves on Jikes RVM
    - exploit compile-time knowledge to customize dynamic type checking code sequence
    - co-design of VM data structures & inline opt code

Disadvantages of JIT/VM Interdependency

- Leakage of implementation details
  - JIT implementation dependent on details VM and vice versa
  - Often performance critical code, so complete abstraction is not always possible
- Maintain JIT/VM interface
  - Interface is often fairly wide and not explicitly specified
  - Changes require coordination and careful planning
    - JIT and VM often owned by different development teams
- Hard to build a JIT that can be plugged into multiple VMs
  - Can be done, but requires discipline and careful design

Course Outline

1. Background
2. Engineering a JIT Compiler
3. **Adaptive Optimization**
    - **Selective Optimization**
    - Design: profiling and recompilation
    - Case studies: Jikes RVM and IBM DK for Java
    - Understanding system behavior
    - Other issues
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

Selective Optimization

- Hypothesis: most execution is spent in a small pct. of methods
  - 90/10 (or 80/20) rule
- Idea: use two execution strategies
  1. **Unoptimized**: interpreter or non-optimizing compiler
  2. **Optimized**: Full-fledged optimizing compiler
- Strategy
  - Use unoptimized execution initially for all methods
  - Profile application to find "hot" subset of methods
    - Optimize this subset
    - Often many times

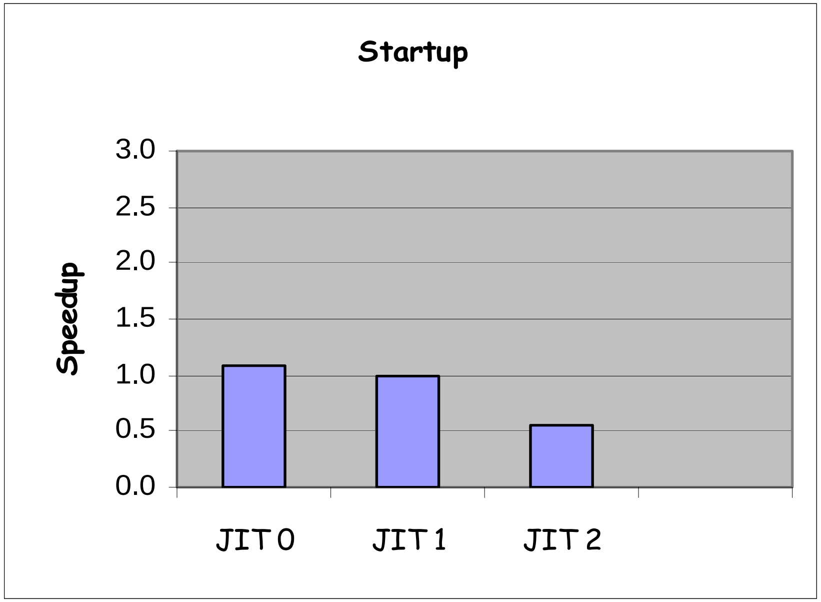
Selective Optimization Examples

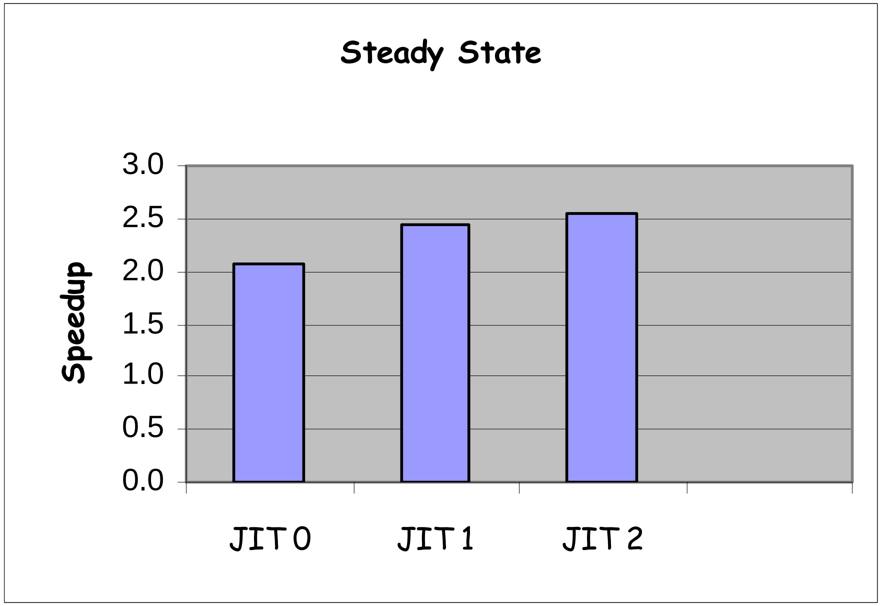
- Adaptive Fortran: interpreter + 2 compilers
- Self'93: non-optimizing + optimizing compilers
- JVMs
  - Interpreter + compilers: Sun's HotSpot, IBM DK for Java, IBM's J9
  - Multiple compilers: Jikes RVM, Intel's Judo/ORP, BEA's JRockit

- CLR
  - only 1 runtime compiler, i.e., a classic JIT
    - But, also use ahead-of-time (AOT) compilation (NGEN)

Selective Optimization Effectiveness:

Jikes RVM, [Arnold et al.,TR Nov'04]



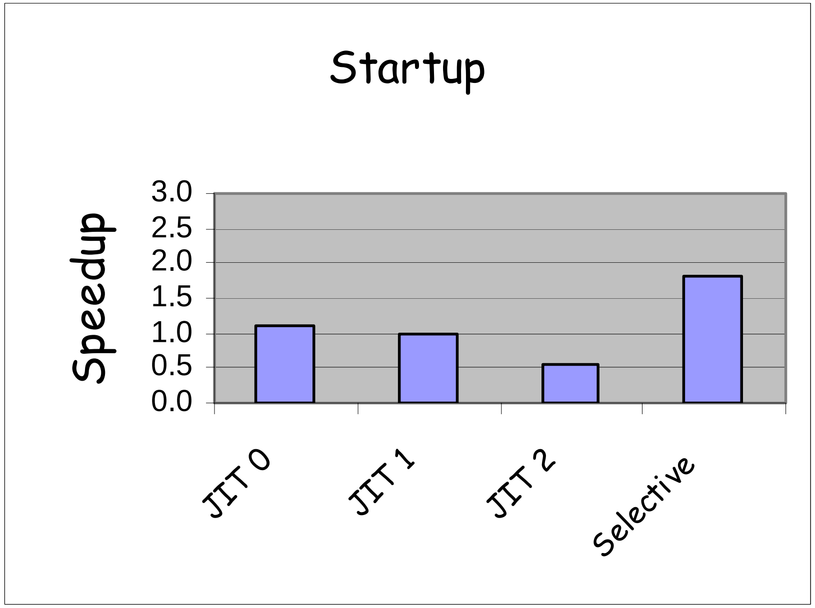


Geometric mean of 12 benchmarks
run with 2 different size inputs
(SPECjvm98, SPECjbb2000, etc.)

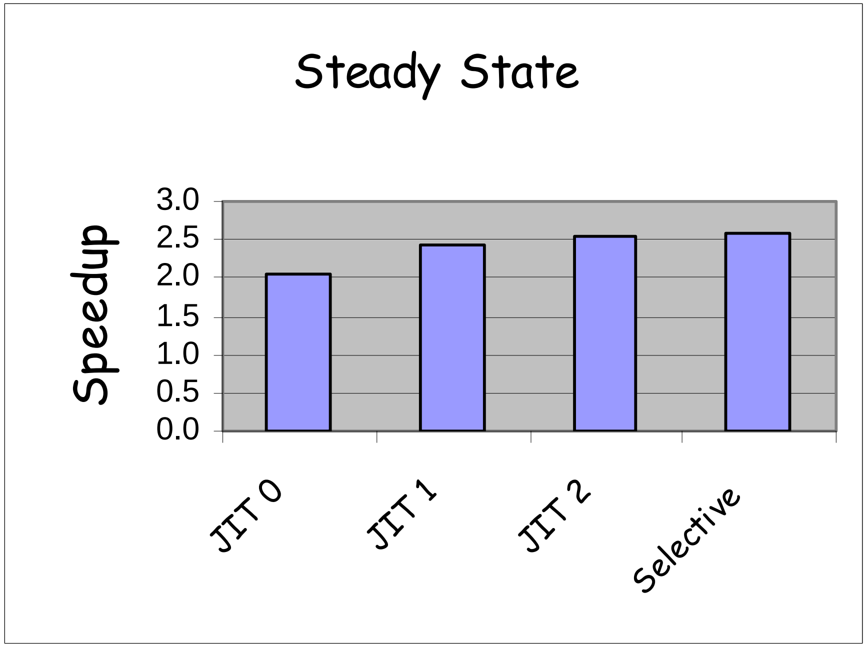
Geometric mean of 9 benchmarks
Best of 20 iterations, default/big inputs
(SPECjvm98, SPECjbb2000, ipsixql)

Selective Optimization Effectiveness:

Jikes RVM, [Arnold et al.,TR Nov'04]



Geometric mean of 12 benchmarks
run with 2 different size inputs
(SPECjvm98, SPECjbb2000, etc.)



Geometric mean of 9 benchmarks
Best of 20 iterations, default/big inputs
(SPECjvm98, SPECjbb2000, ipsixql)

Designing an Adaptive Optimization System

- What is the system architecture for implementing selective optimization?
- What is the mechanism (profiling) and policy for driving recompilation?
- How effective are existing systems?

Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
   - Selective optimization
   - **Design: profiling and recompilation**
   - Case studies: Jikes RVM and IBM DK for Java
   - Understanding system behavior
   - Other issues
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

Profiling: How to Find Candidates for Optimization

- Counters
- Call Stack Sampling
- Combinations

How to Find Candidates for Optimization: Counters

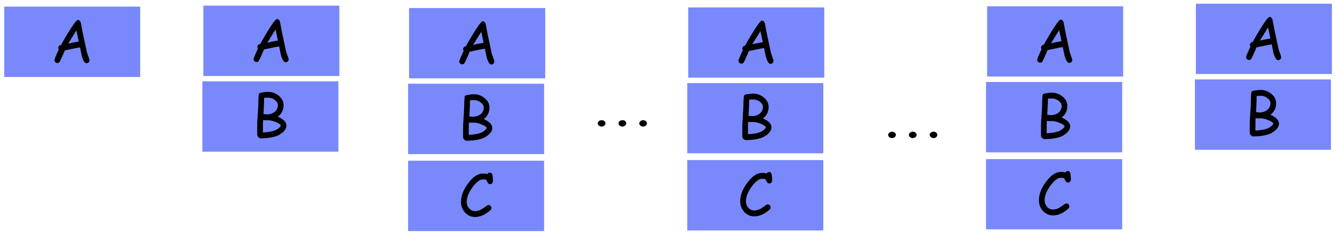
- Insert method-specific counter on method entry and loop back edge
- Counts how often a method is called
  - approximates how much time is spent in a method
- Very popular approach: Self, HotSpot
- Issues: overhead for incrementing counter can be significant
  - Not present in optimized code

```
foo ( … ) {
    fooCounter++;
    if (fooCounter > Threshold) {
        recompile( … );
    }
    . . .

}
```

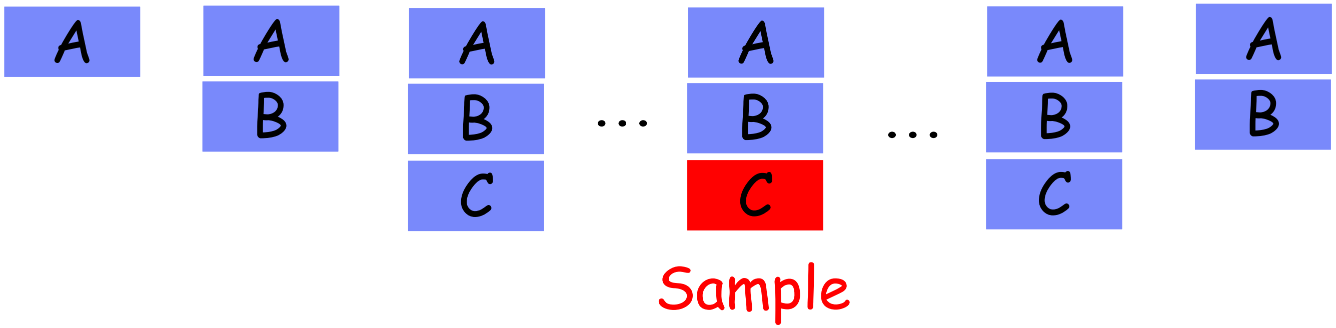
How to Find Candidates for Optimization: Call Stack Sampling

- Periodically record which method(s) are on the call stack
- Approximates amount of time spent in each method
- Does not necessarily need to be compiled into the code
  - Ex. Jikes RVM, JRocket
- Issues: timer-based sampling is not deterministic



How to Find Candidates for Optimization: Call Stack Sampling

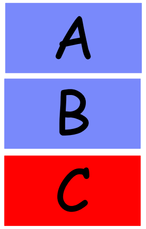
- Periodically record which method(s) are on the call stack
- Approximates amount of time spent in each method
- Does not necessarily need to be compiled into the code
  - Ex. Jikes RVM, JRocket
- Issues: timer-based sampling is not deterministic



How to Find Candidates for Optimization

- Combinations
  - Use counters initially and sampling later on
  - Ex) IBM DK for Java, J9

```
foo ( … ) {
    fooCounter++;
    if (fooCounter > Threshold) {
        recompile( … );
    }
    . . .
}
```



A

B

C

Recompilation Policies: Which Candidates to Optimize?

- Problem: given optimization candidates, which ones should be optimized?
- Counters
  1. Optimize method that surpasses threshold
     - Simple, but hard to tune, doesn't consider context
  2. Optimize method on the call stack based on inlining policies (Self, HotSpot)
     - Addresses context issue
- Call Stack Sampling
  1. Optimize all methods that are sampled
     - Simple, but doesn't consider frequency of sampled methods
  2. Use Cost/benefit model (Jikes RVM)
     - Seemingly complicated, but easy to engineer
     - Maintenance free
     - Naturally supports multiple optimization levels

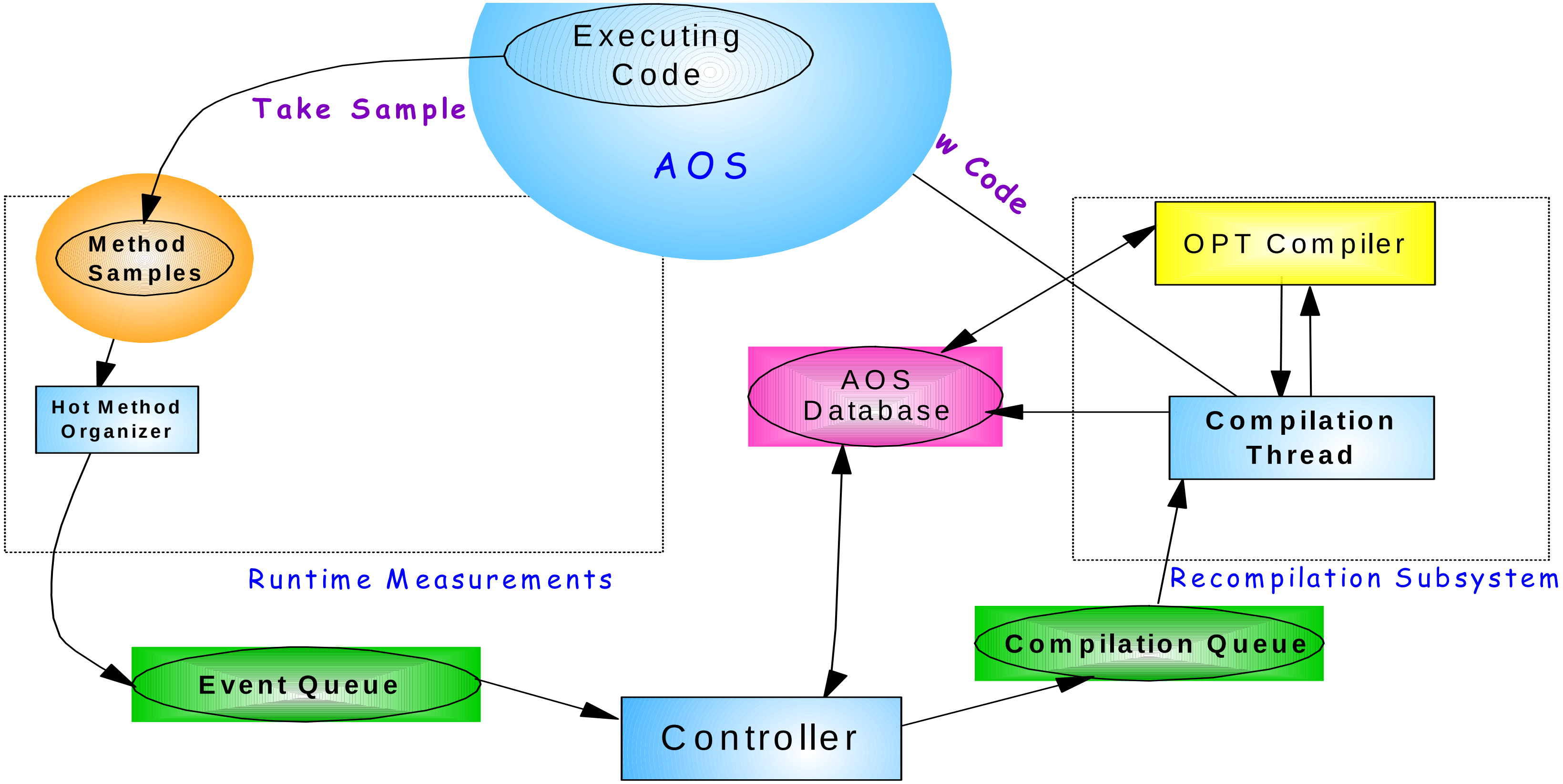
Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
    - Selective optimization
    - Design: profiling and recompilation
    - **Case studies: Jikes RVM and IBM DK for Java**
    - Understanding system behavior
    - Other issues
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

Case Studies

- Jikes RVM [Arnold et al. '00]
- IBM DK for Java [Suganuma et al. '01, '05]

Case Study 1: Jikes RVM Architecture [Arnold et al. '00]



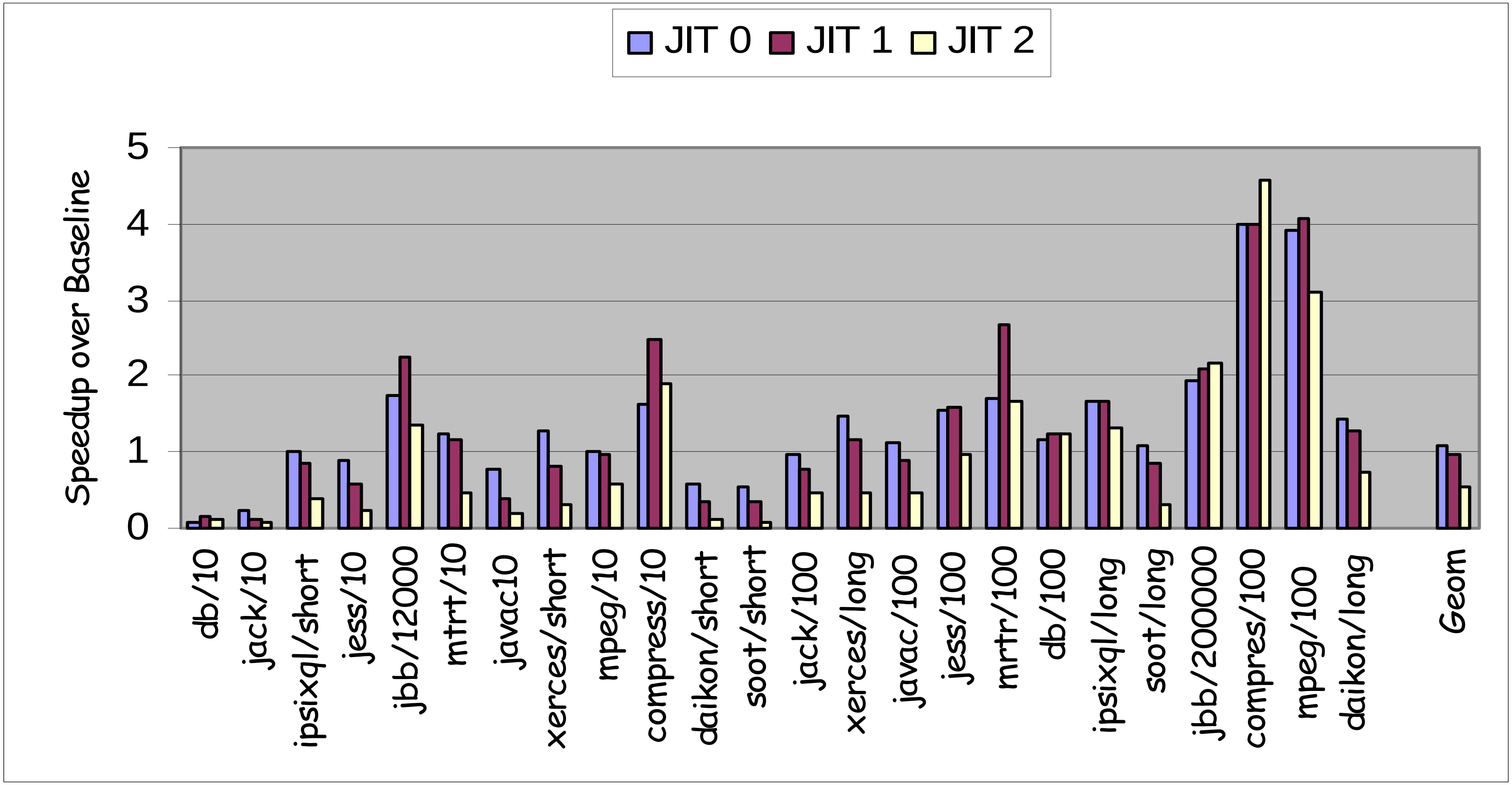
Samples occur at taken yield points (approx 100/sec)

Organizer thread communicates sampled methods to controller

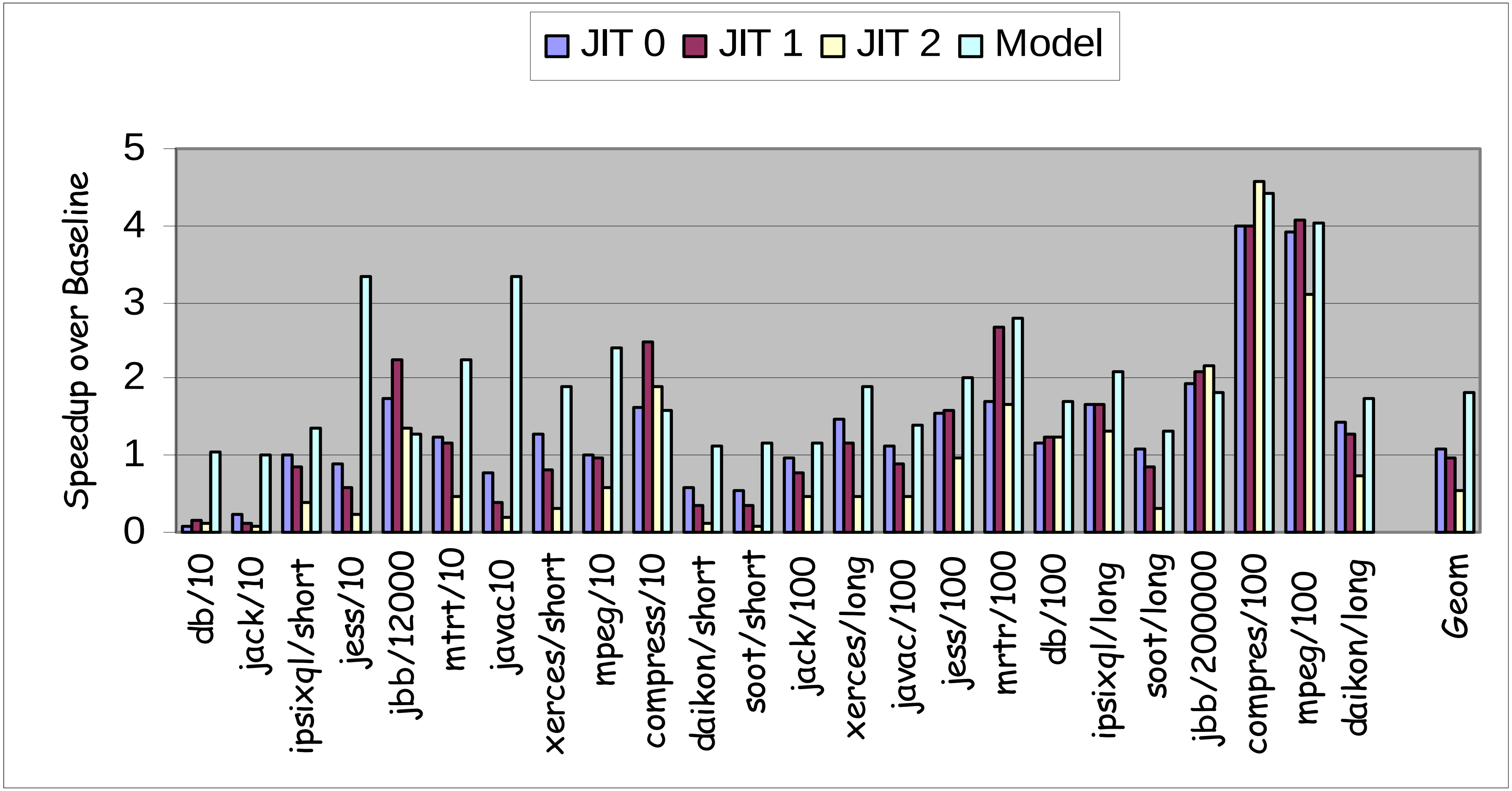
Jikes RVM: Recompilation Policy – Cost/Benefit Model

- Define
  - cur, current opt level for method m
  - Exe(j), expected future execution time at level j
  - Comp(j), compilation cost at opt level j
- Choose j > cur that minimizes Exe(j) + Comp(j)
- If Exe(j) + Comp(j) < Exe(cur) recompile at level j
- Assumptions
  - Sample data determines how long a method has executed
  - Method will execute as much in the future as it has in the past
  - Compilation cost and speedup are offline averages

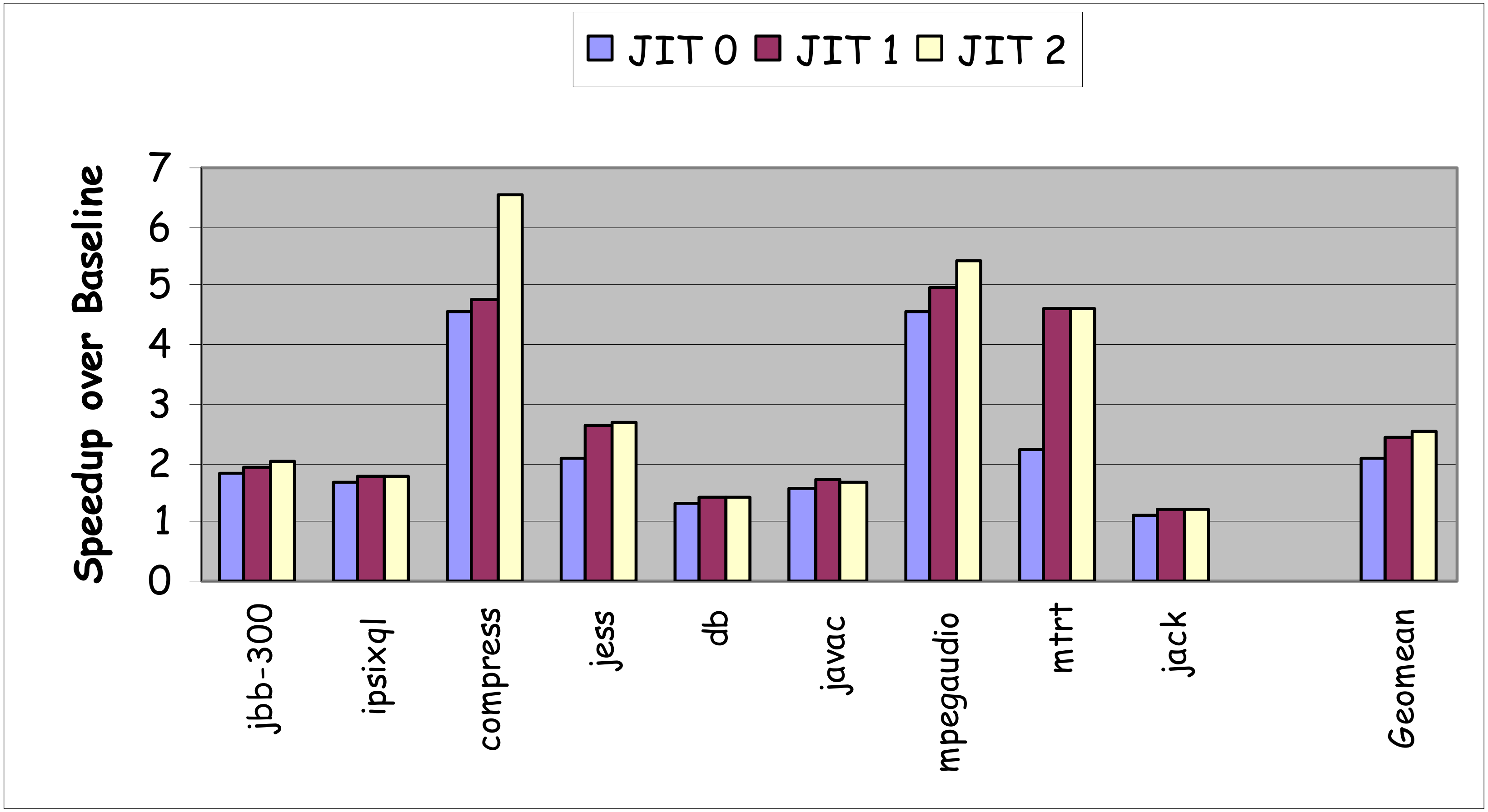
Short-running Programs: Jikes RVM



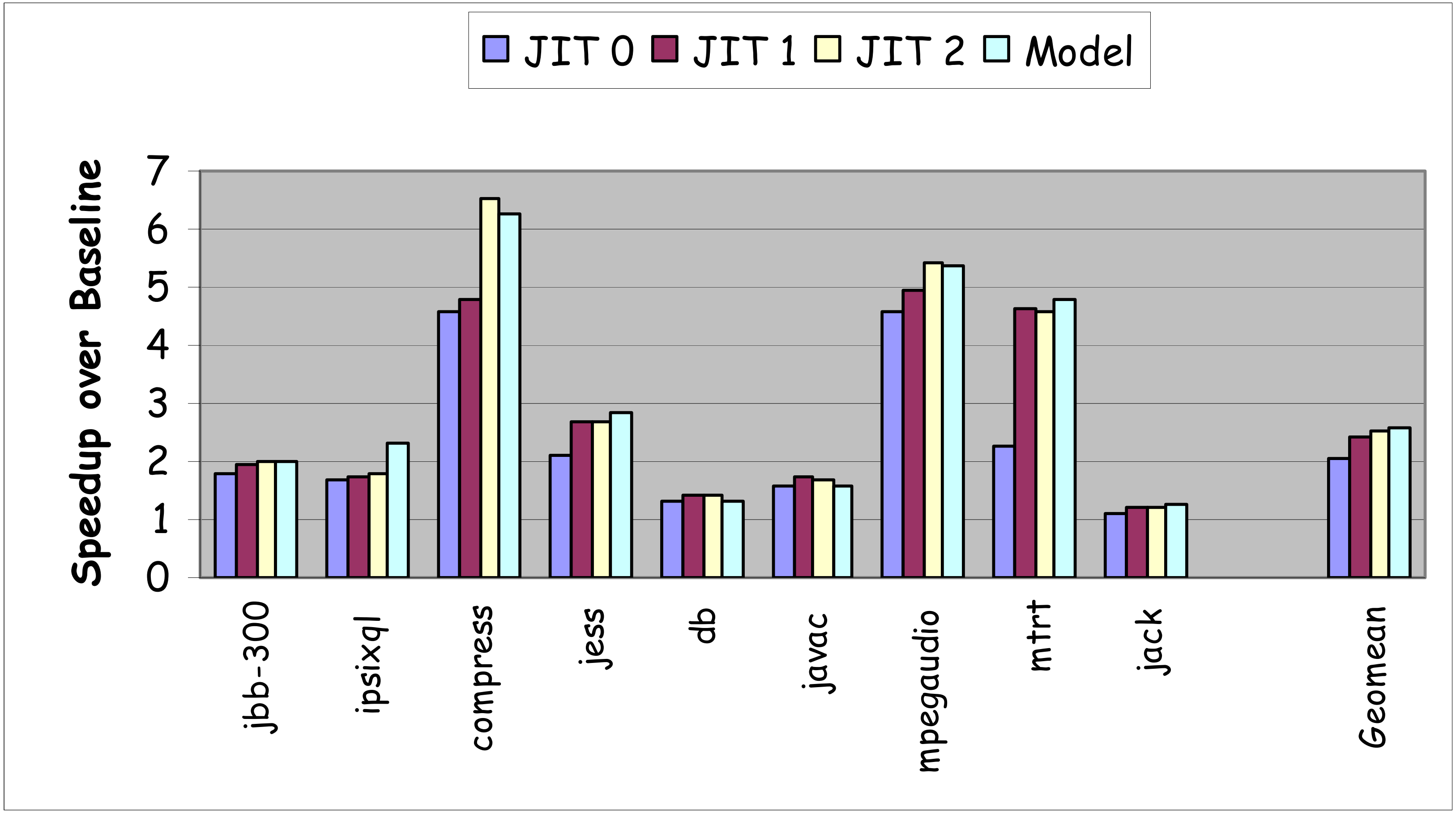
Short-running Programs: Jikes RVM



Steady State: Jikes RVM



Steady State: Jikes RVM, no FDO (Mar '04)

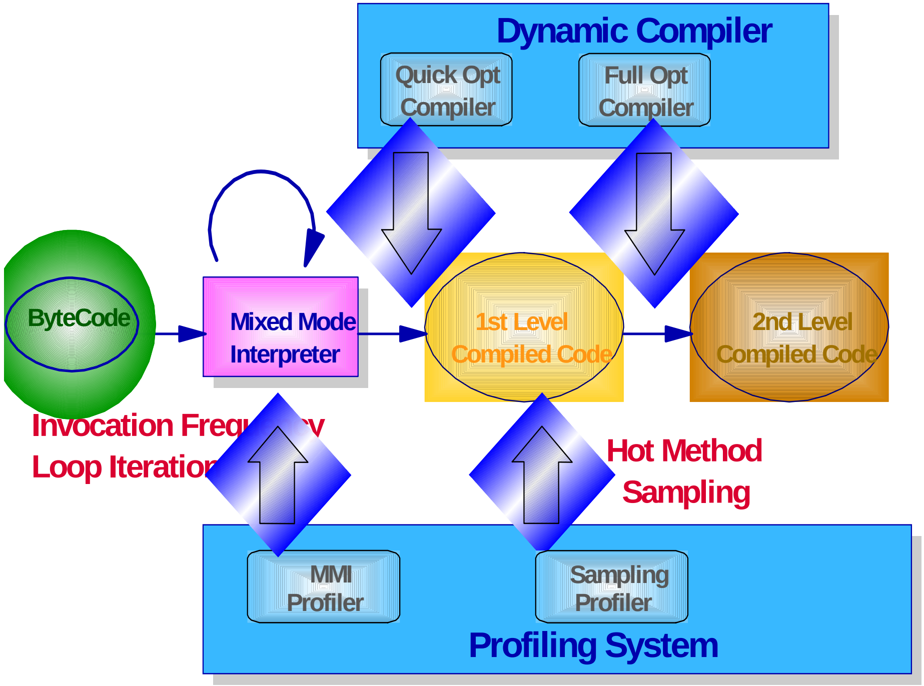


Case Study 2: IBM DK for Java [Suganuma et al. '01, '05]

Execution Levels (excluding Specialization)

- MMI (Mixed Mode Interpreter)
  - Fast interpreter implemented in assembler
- Quick compilation
  - Reduced set of optimizations for fast compilation, little inlining
- Full compilation
  - Full optimizations only for selected hot methods

- Methods can progress sequentially through the levels



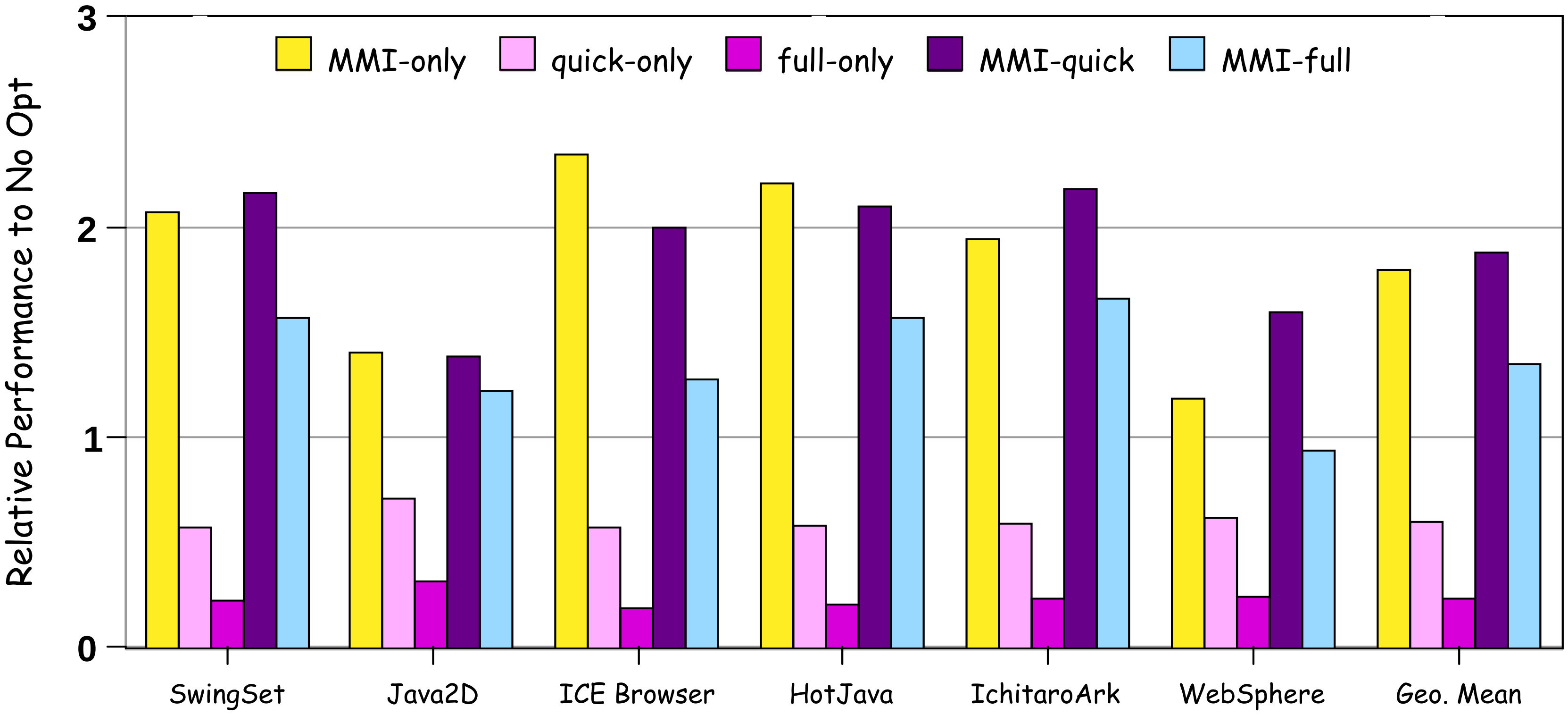
Profile Collection

- MMI Profiler (Counter Based)
  - Invocation frequency and loop iteration
- Sampling Profiler
  - Lightweight for operating during the entire execution
  - Only monitors compiled methods
  - Maintains list of hot methods and calling relationships among hot methods
- MMI also collects branch frequencies for FDO

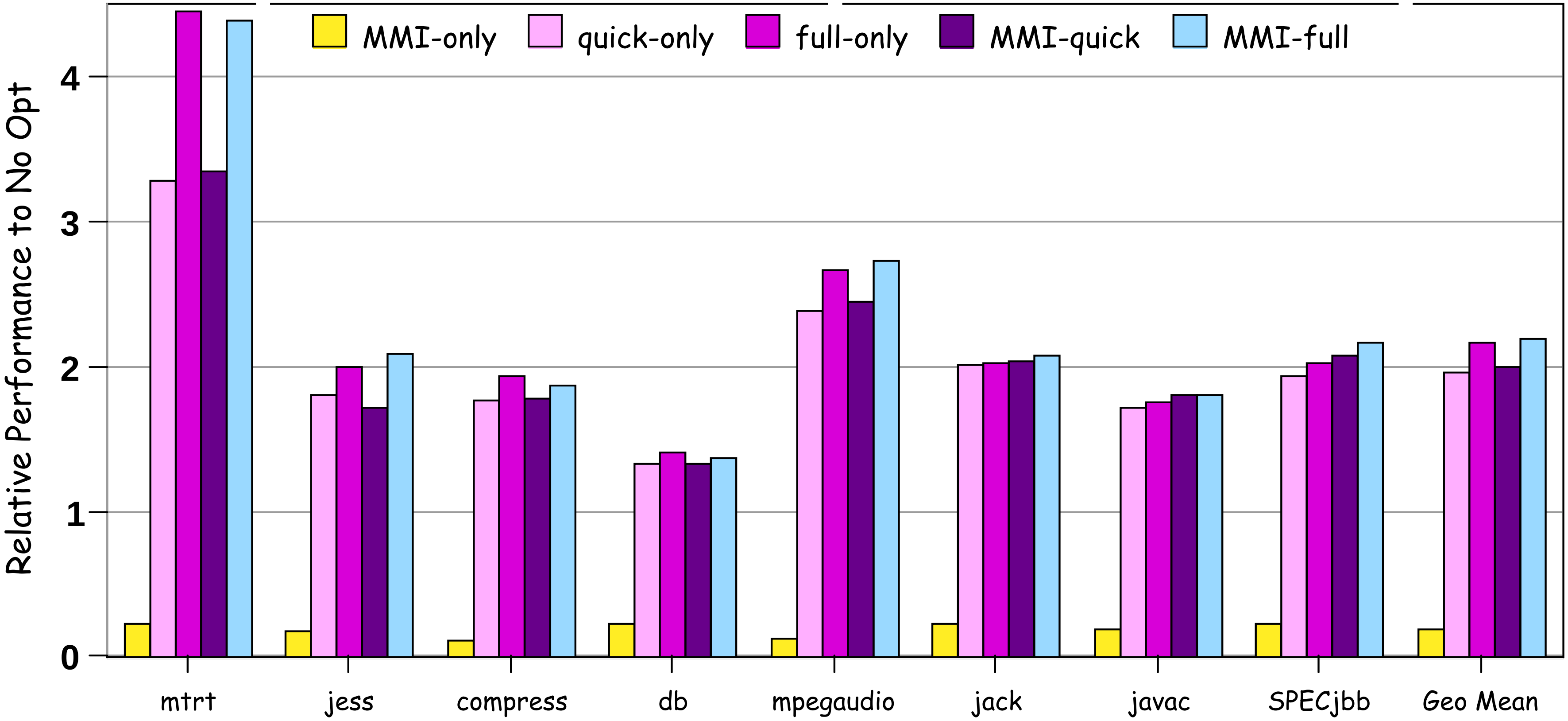
Recompilation Policy

- Methods are promoted sequentially through the levels
- MMI -> Quick
  - Based on loop and invocation counts with special treatment for certain types of loops
- Quick -> Full
  - Based on sampling profiler
  - Roots of call graphs are recompiled with inlining directives
    - Inspired by Self'93

Startup: IBM DK for Java, no Specialization [Suganuma et al. '01]



Steady State: IBM DK for Java, no Specialization [Suganuma et al. '01]

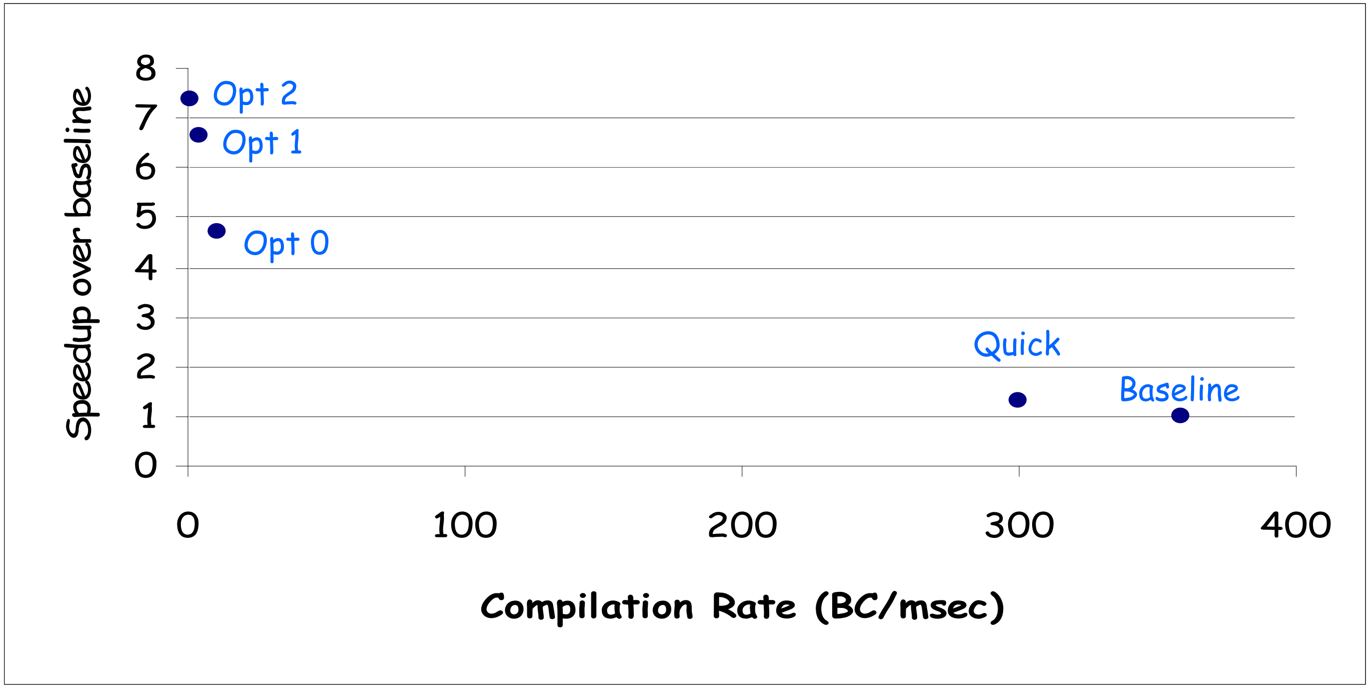


But the world is not always simple

- Modern programs execute a large number of methods
- SPECjappserver, Mark Stoodley (IBM) MRE'05
  - executes > 10,000 methods
  - No single "hot spot"
    - Hottest method may be <1% of total execution time
  - 90/10 rule may still apply
    - But 10% of 10,000 is 1,000 (warm) methods
- Eclipse startup, IBM J9 VM

Workspace	Running Time	Number of Methods		
		Exe.	Optimized	Highest Level
Empty	5.8 secs	10,499	740 (7.1%)	4 (0.04%)
Eclipse source	18.2 secs	18,960	2,169 (11.4%)	21 (0.11%)

Example: Jikes RVM Compilers on AIX/PPC



Both efficiency and code quality of optimization are relevant

- Improving the efficiency of optimization has value
- Improving code quality has value
  - Even if expensive, can likely be incorporated via selective optimization

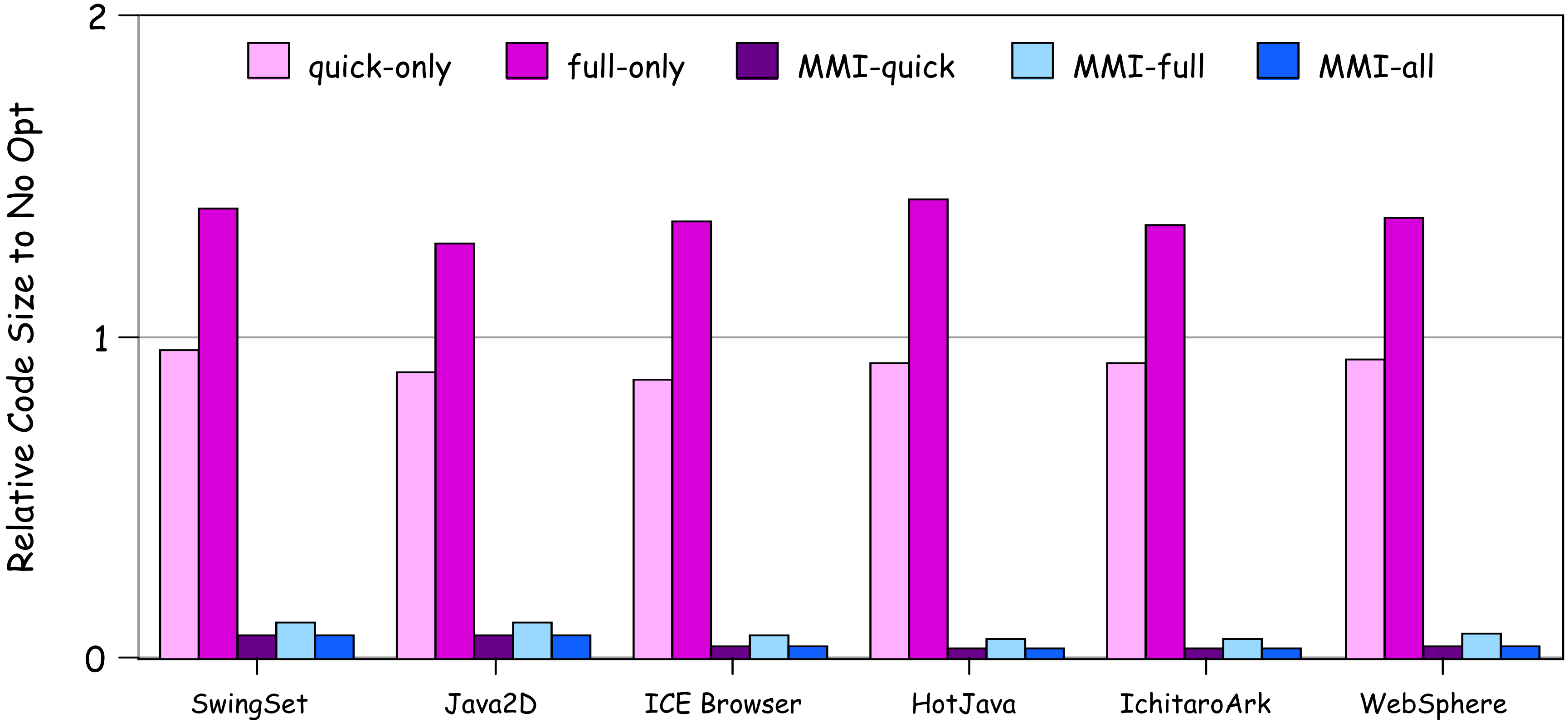
Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
    - Selective optimization
    - Design: profiling and recompilation
    - Case studies: Jikes RVM and IBM DK for Java
    - **Understanding system behavior**
    - Other issues
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

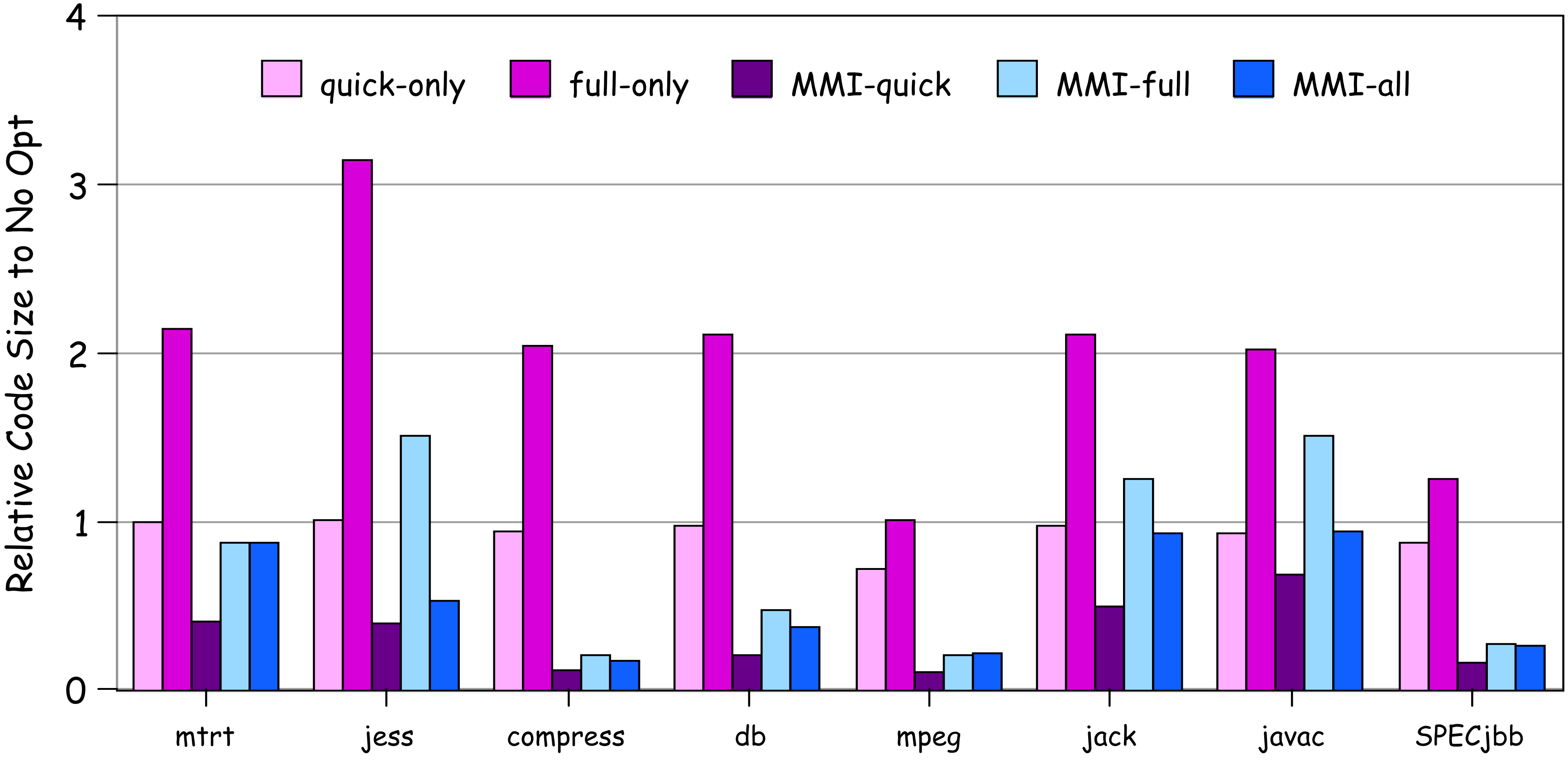
Understanding System Behavior

- Code size usage (IBM DK for Java)
- Execution time overhead (Jikes RVM)
- Recompilation information
  - Pct/total methods recompiled (Jikes RVM)
  - Activity over time (Both)

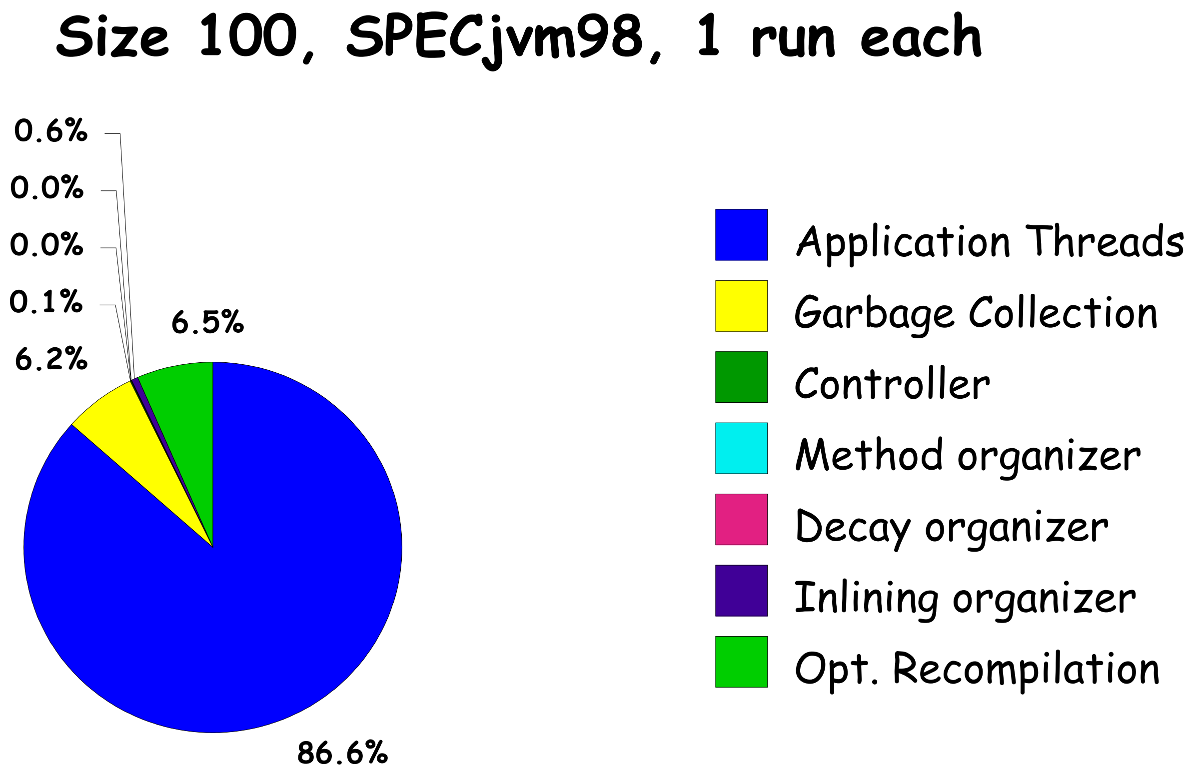
Code Size Comparison, startup: IBM DK for Java



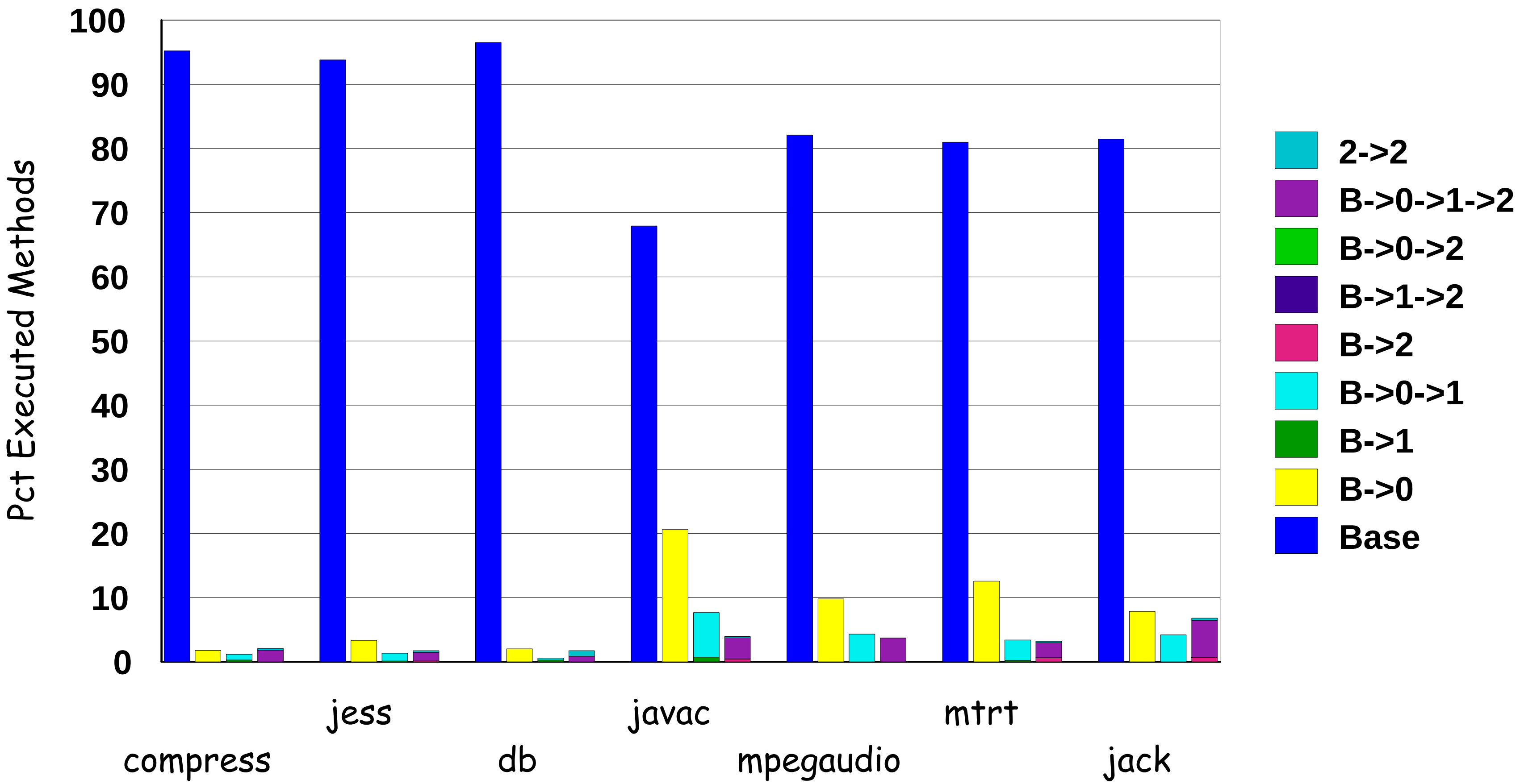
Code Size Comparison, steady state: IBM DK for Java



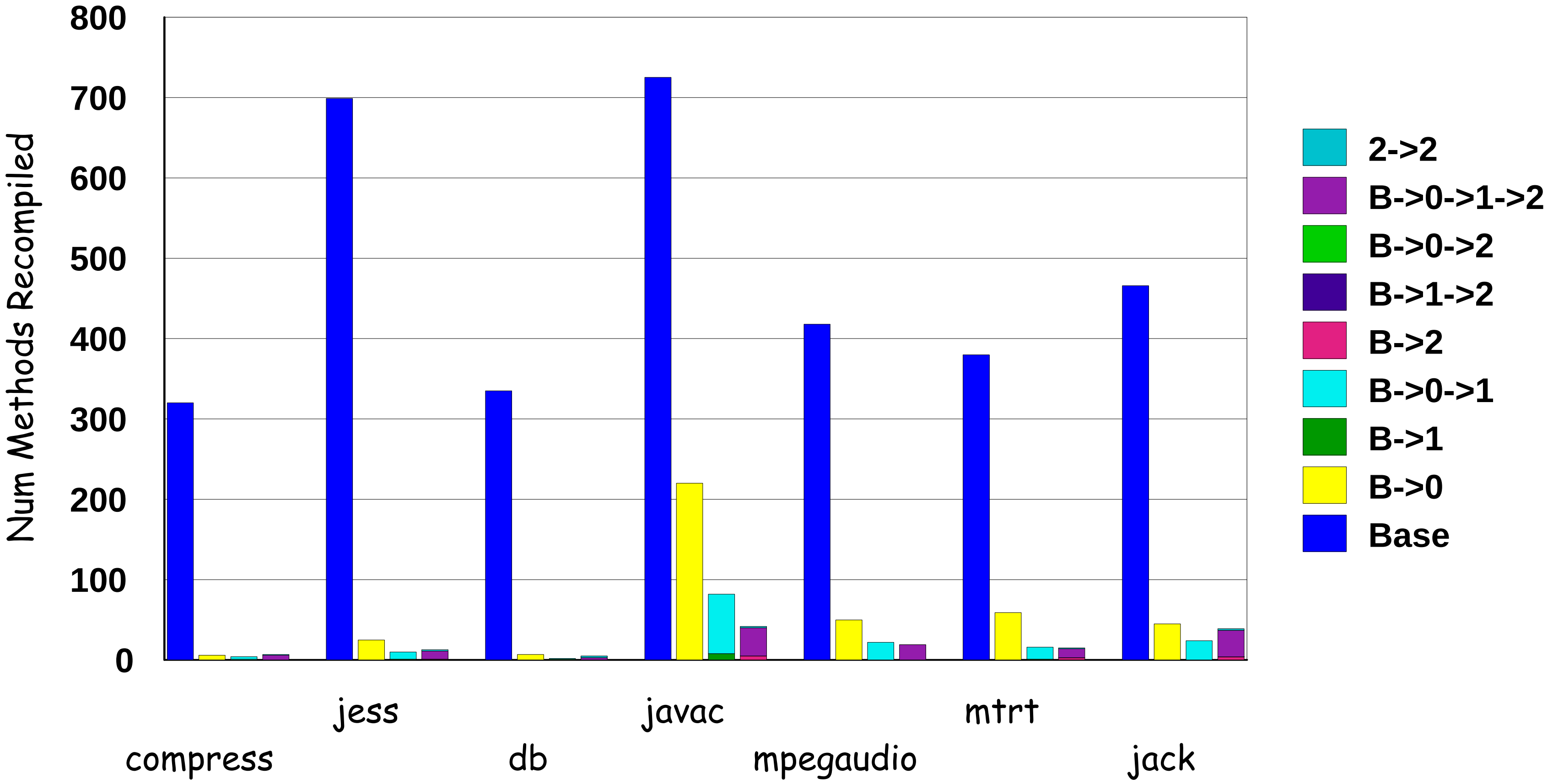
Execution Profile: Jikes RVM (Jul '02)



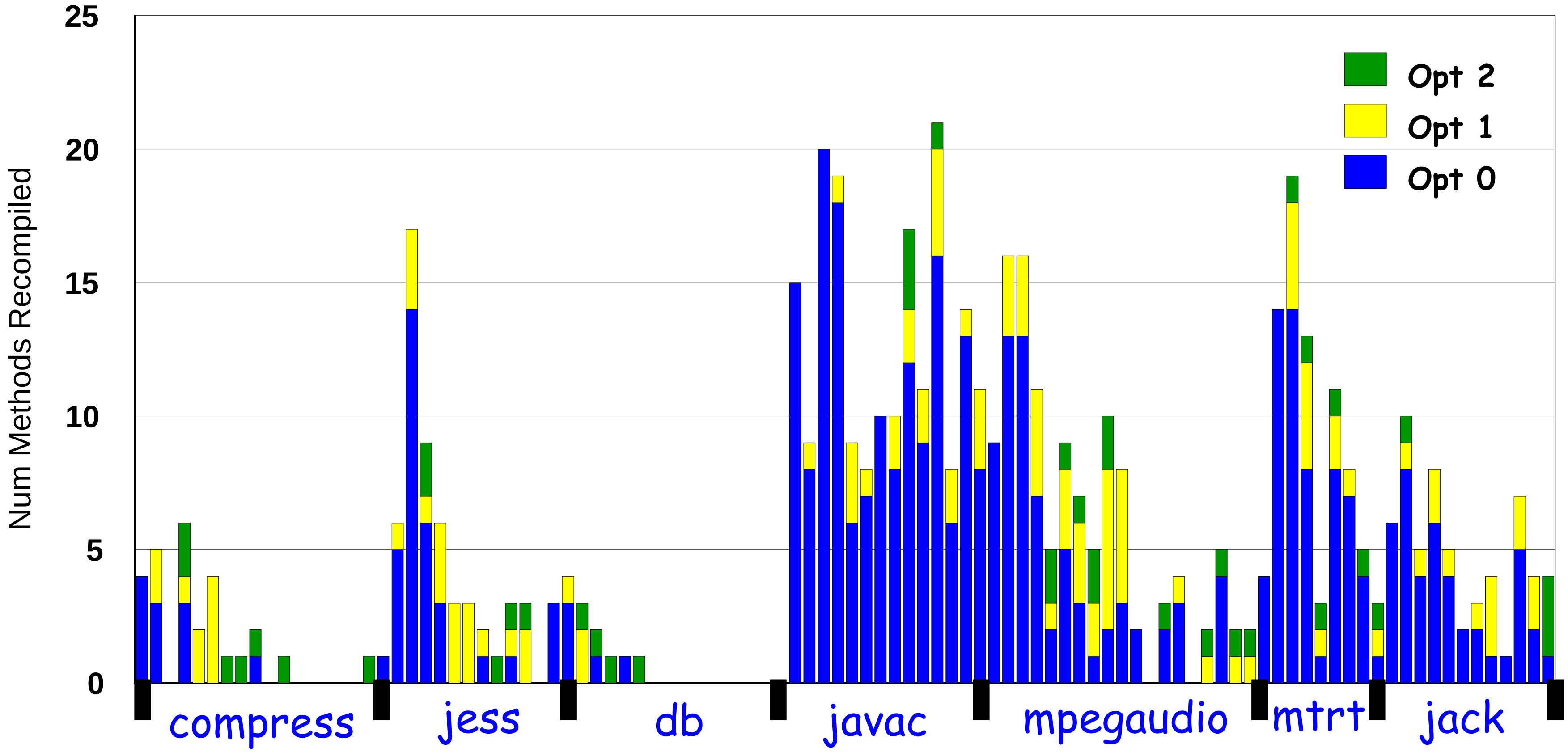
Recomp. Decisions, 20 iterations for each benchmark Jikes RVM



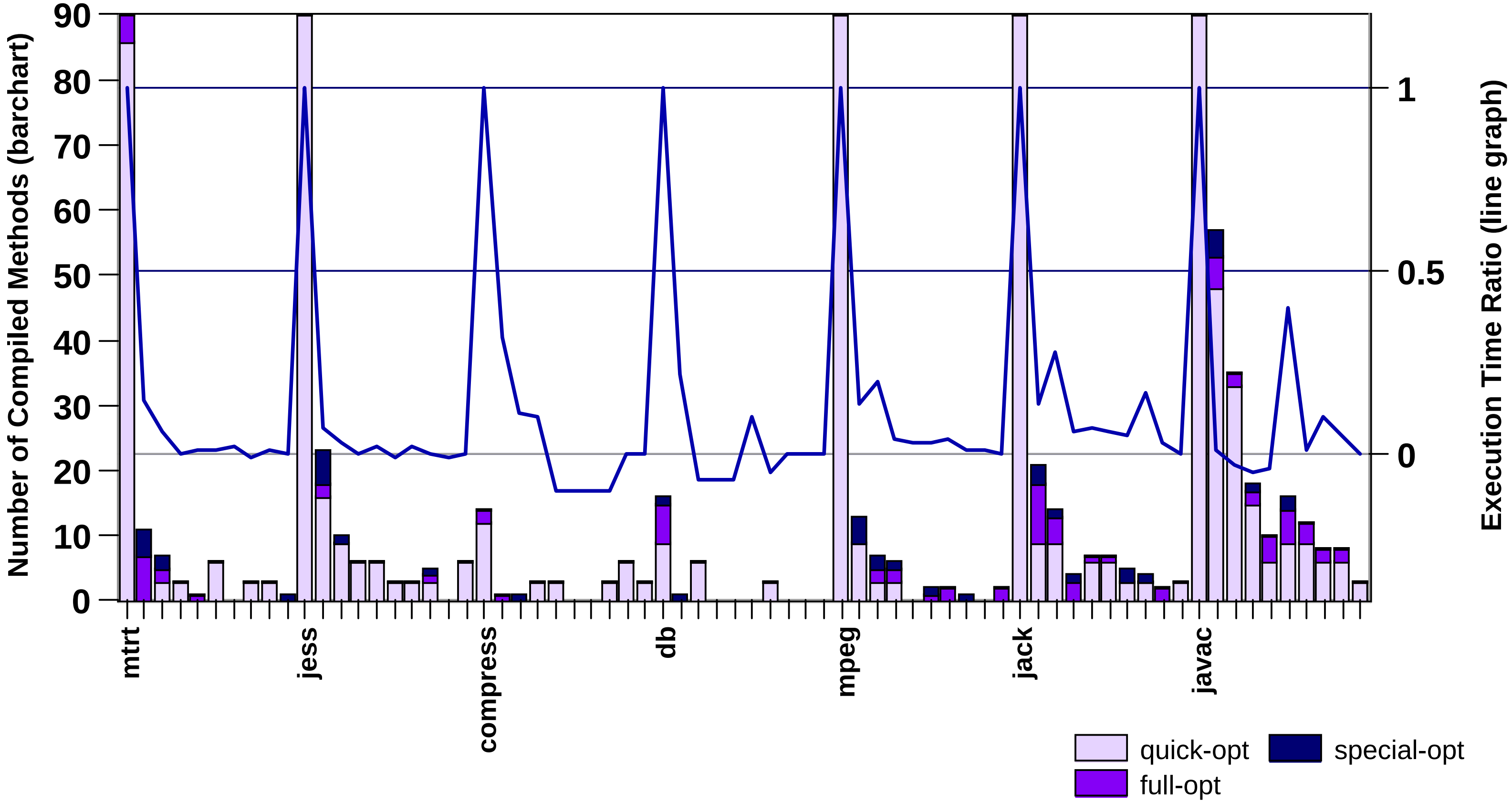
Recomp. Decisions, 20 iterations for each benchmark Jikes RVM



Recompilation Activity: Jikes RVM (Jul '02)



Recompilation Activity (IBM DK for Java)



Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
    - Selective optimization
    - Design: profiling and recompilation
    - Case studies: Jikes RVM and IBM DK for Java
    - Understanding system behavior
    - **Other issues**
4. Feedback-Directed and Speculative Optimizations
5. Summing Up and Looking Forward

Research Issues for Adaptive Optimization (1/2)

- Tuning thresholds is a problem
  - Threshold values often turn out to be bad later on
  - Dealing with combined counter and sample data
- Pause times
  - Model optimizes throughput, ignores pauses
    - After running for an hour, may suggest massive compilations
- Synchronous vs. asynchronous recompilation
  - Is optimization performed in the background, or is the application suspended during compilation?
  - Exploit idle CPU's
    - Dozens of compilations in parallel (Azul Systems)
- Static or dynamic view of profile data
  - Is profile data packaged or used in flight during compilation?

Research Issues for Adaptive Optimization (2/2)

- Skipping optimization levels
  - When to do it?
  - Better ways to predict how long method will run?
- Handling programs with “flat” profiles
  - Use partial method compilation?
- Handling code space
  - Do we need to budget recompilation?
- Responsiveness of installing new compiled code
  - Stack rewriting, code patching, etc.
- Reliability
  - Repeatability
  - Counters have advantages, and disadvantages
- Can we save information for future runs?
  - More details to follow

Learning From a Previous Run

Q: Why throw away everything a VM has learned just because the program has ended?

A: Several approaches exist

- Quicksilver [OOPSLA'00]
  - Save the compiled code for a subsequent execution
  - Issue: need to deal with security issue, phase changes
- Krintz & Calder [PLDI'01, CGO'03]
  - Add annotation to classfiles for important methods
  - Issue: annotations are independent from online recompilation strategy
- Arnold et al. [OOPSLA'05]
  - Details to follow

Arnold, Welc, Rajan [OOPSLA'05]

- JVMs apply compilation at runtime
  - Better predictions of method running time allows better use of JIT compiler

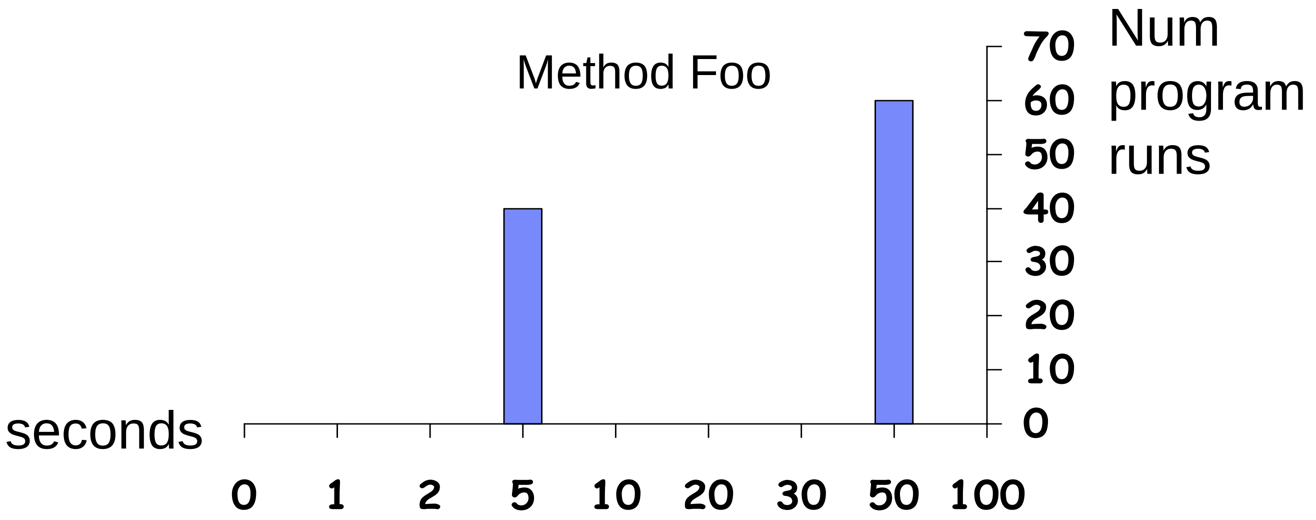
- Database stores method execution patterns from multiple runs
  - Optimization strategies constructed based on these patterns
    - Read by JVM at startup, if exists

- Average startup improvement

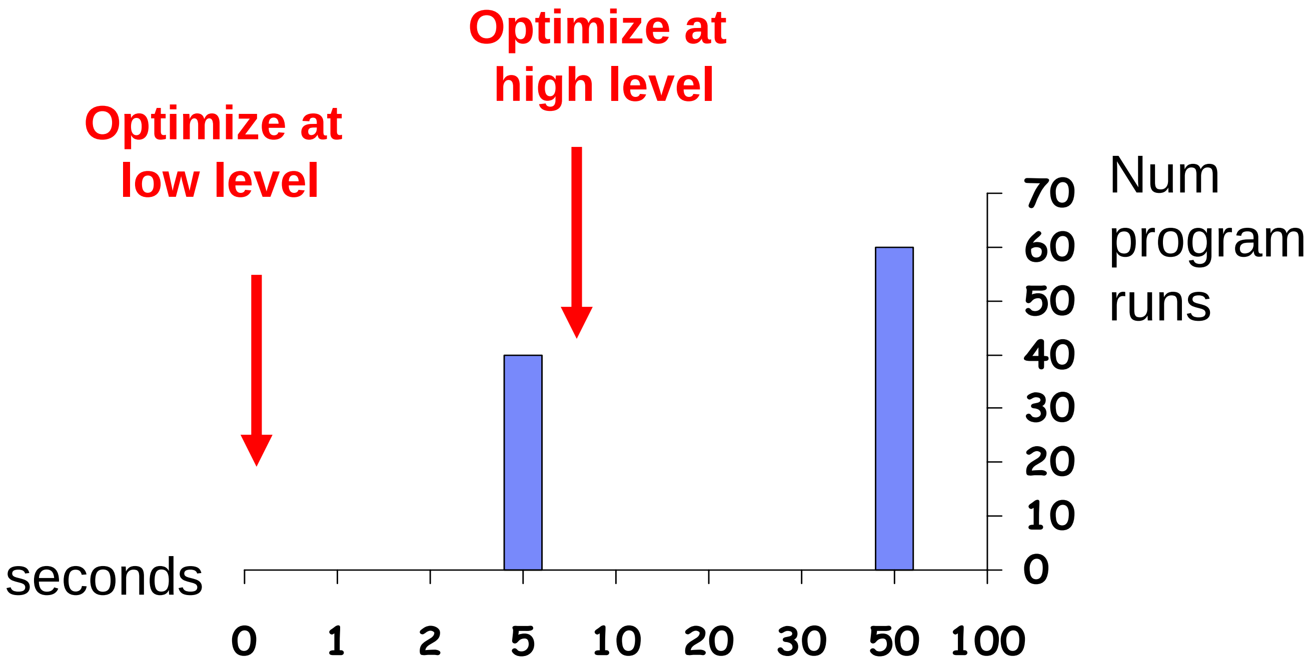
  8 – 16% depending on execution scenario

Profile Repository: Histogram of Method Runtimes

- For each (hot) method in the program
  - Record how much time spent in the method during each program execution
  - After each run, update a histogram of run times
  - Example: method Foo
    - Ran program 100 times
    - In 40 program runs, Foo executed for 5 seconds
    - In 60 runs, Foo executed for 50 seconds



Profile Repository: Histogram of Method Runtimes



Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
4. **Feedback-Directed and Speculative Optimizations**
    - **Gathering profile information**
    - Exploiting profile information in a JIT
        - Feedback-directed optimizations
        - Aggressive speculation and invalidation
    - Exploiting profile information in a VM
5. Summing Up and Looking Forward

Feedback-Directed Optimization (FDO)

- Exploit information gathered at runtime to optimize execution
  - "selective optimization": **what** to optimize
  - "FDO" : **how** to optimize
    - Similar to offline profile-guided optimization
    - Only requires 1 run!
- Advantages of FDO [Smith'00]
  - Can exploit dynamic information that cannot be inferred statically
  - System can change and revert decisions when conditions change
  - Runtime binding has advantages
- Performed in many systems
  - Eg, Jikes RVM, 10% improvement using FDO
    - Using basic block frequencies and call edge profiles
- Many opportunities to use profile info during various compiler phases
  - Almost any heuristic-based decision can be informed by profile data
    - Inlining, code layout, multiversioning, register allocation, global code motion, exception handling optimizations, loop unrolling, speculative stack allocation, software prefetching

Issues in Gathering Profile Data

1. What data do you collect?
2. How do you collect it?
3. When do you collect it?

Issue 1: What data do you collect?

- What data do you collect?
  - Branch outcomes
  - parameter values
  - loads and stores
  - etc.
- Overhead issues
  - cost to collect, store, and use data

Issue 2: How do you collect the data?

- Program instrumentation
  - e.g. basic block counters, value profiling
- Sampling [Whaley, JavaGrande'00; Arnold&Sweeney TR'00; Arnold&Grove, CGO'05; Zhuang et al. PLDI'06]
  - e.g. sample method running, call stack at context switch
- Hybrid: [Arnold&Ryder, PLDI'01]
  - combine sampling and instrumentation
- Runtime service monitors
  [Deutsch&Schiffman, POPL'84,Hölzle et al., ECOOP'91; Kawachiya et al., OOPSLA'02; Jones&Lins'96]
  - e.g. dispatch tables, synchronization services, GC
- Hardware performance monitors: [Ammons et al. PLDI'97; Adl-Tabatabai et al., PLDI'04]
  - e.g. drive selective optimization, suggest locality improvements

Issue 3: When do you collect the data?

When do you collect the data?

- During different execution modes (interpreter or JIT)
  - e.g. Profile branches during interpetation
  - e.g. Add instrumentation during execution of JITed code

- During different application phases (early, steady state, etc.)
  - Profile during initial execution to use during steady state execution
  - Profile during steady state to predict steady state

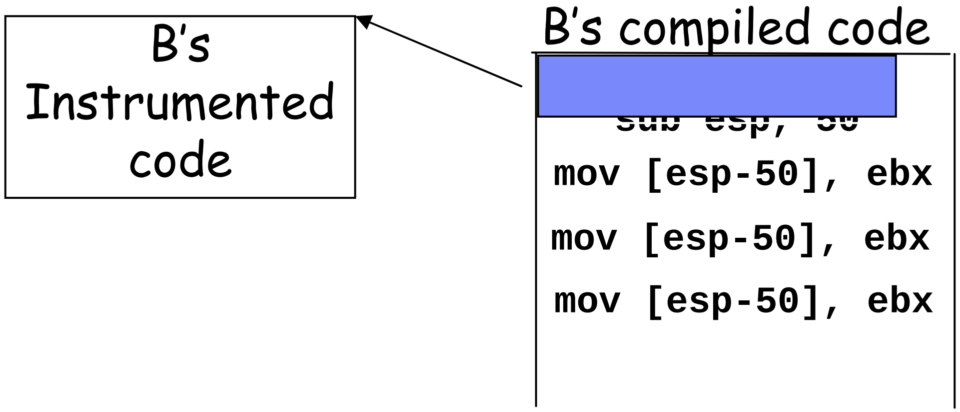
- Issues: overhead vs accuracy of profile data

Common Approaches in VMs

- Most VMs perform profiling during initial execution (interpretation or initial compiler)
  - Easy to implement
  - Low-overhead (compared to unoptimized code)
  - Typically branch profiles are gathered
  - Leads to nontrivial FDO improvements
    - 10% for Jikes RVM
- Call stack sampling can be used for optimized code
  - Low overhead
  - Limited profile information
- Some VMs also profile optimized methods using instrumentation
  - Leverages selective optimization strategy
  - Challenge is to keep overhead low (see next 2 slides)

IBM DK Profiler [Suganuma et al '01,'02]

- Sampling
  - Used to identify already compiled methods for re-optimization
- Dynamic instrumentation
  1. *Patch* entry to a method with jump to instrumented version
  2. Run until threshold
     - Time bound
     - Desired quantity of data collected
  3. Undo patch



B's Instrumented code

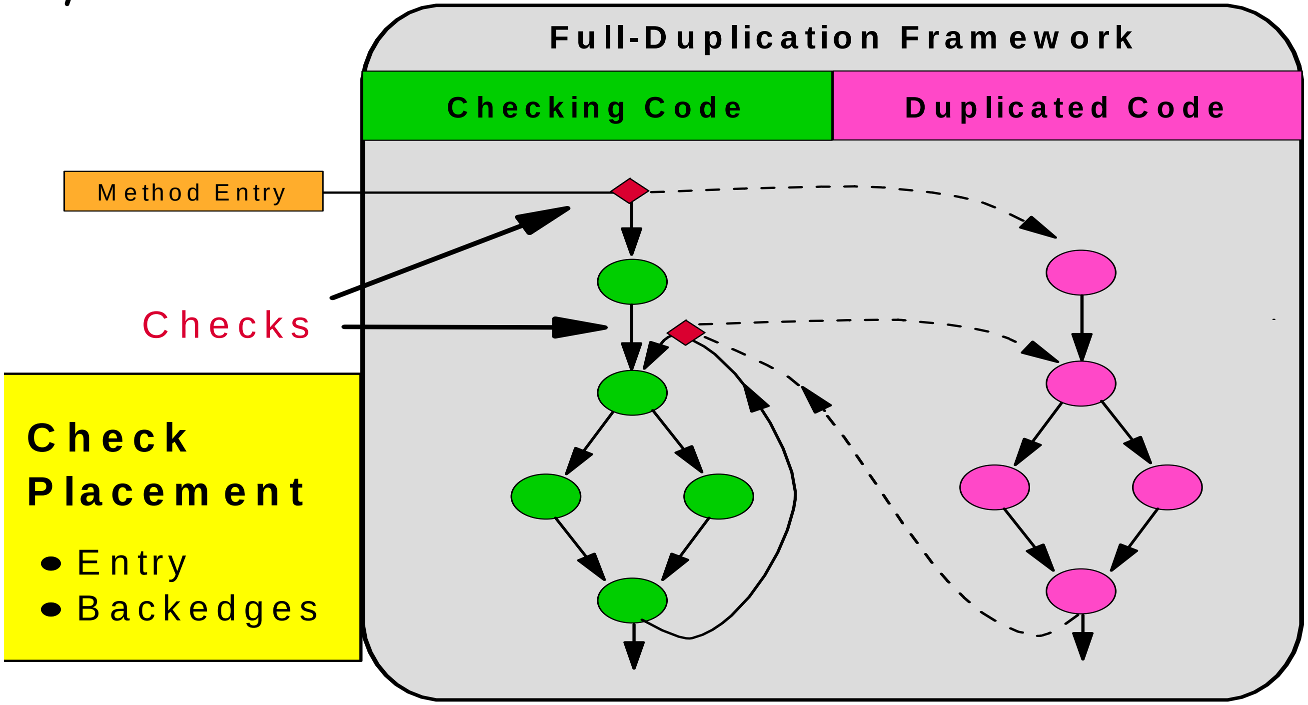
B's compiled code

```
sub esp, 50
mov [esp-50], ebx
mov [esp-50], ebx
mov [esp-50], ebx
```

Arnold-Ryder [PLDI 01]: Full Duplication Profiling

No patching; instead generate two copies of a method

- Execute "fast path" most of the time
- Jump to "slow path" occasionally to collect profile
- Demonstrated low overhead, high accuracy
- Used by J9 and other researchers



Check Placement

- Entry
- Backedges

Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
   - Gathering profile information
   - **Exploiting profile information in a JIT**
     - Feedback-directed optimizations ("3a")
     - Aggressive speculation and invalidation ("3b")
   - Exploiting profile information in a VM
5. Summing Up and Looking Forward

Types of Optimization

1. Ahead of time optimization
   - It is never incorrect, prove for every execution
2. Runtime static optimization
   - Will not require invalidation

     Ex. inlining of final or static methods
3. Speculative optimizations

   Profile, speculate, invalidate if needed

   Two flavors:

   a) True now, but may change

      Ex. class hierarchy analysis-based inlining

   b) True most of the time, but not always

      Ex. speculative inlining with invalidation mechanisms

Current systems perform 2 & 3a, but not much of 3b

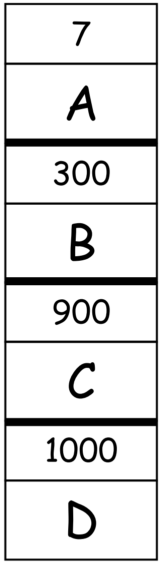
Common FDO Techniques

- Compiler optimizations
  - Inlining
  - Code Layout
  - Multiversioning
  - Potpourri
- Runtime system optimizations
  - Caching
  - Speculative meta-data representations
  - GC Acceleration
  - Locality optimizations

Fully Automatic Profile-Directed Inlining

Example: SELF-93 [Hölzle&Ungar'94]

- Profile-directed inlining integrated with sampling-based recompilation
- When sampling counter triggered, crawl up call stack to find "root" method of inline sequence



7

A

300

B

900

C

1000

D

- D trips counter threshold
- Crawl up stack, examine counters
- Recompile B and inline C and D

Fully Automatic Profile-Directed Inlining

Example: IBM DK for Java [Suganuma et al. '02]

- Always inline "tiny" methods (e.g. getters)
- Use dynamic instrumentation to collect call site distribution
  - Determine the most frequently called sites in "hot" methods
- Constructs partial dynamic call graph of "hot" call edges
- Inlining database to avoid performance perturbation

- Experimental conclusion
  - use static heuristics only for small size methods
  - inline medium- and bigger only based on profile data

Inlining Trials in SELF [Dean and Chambers 94]

Problem: Estimating inlining effect on optimization is hard

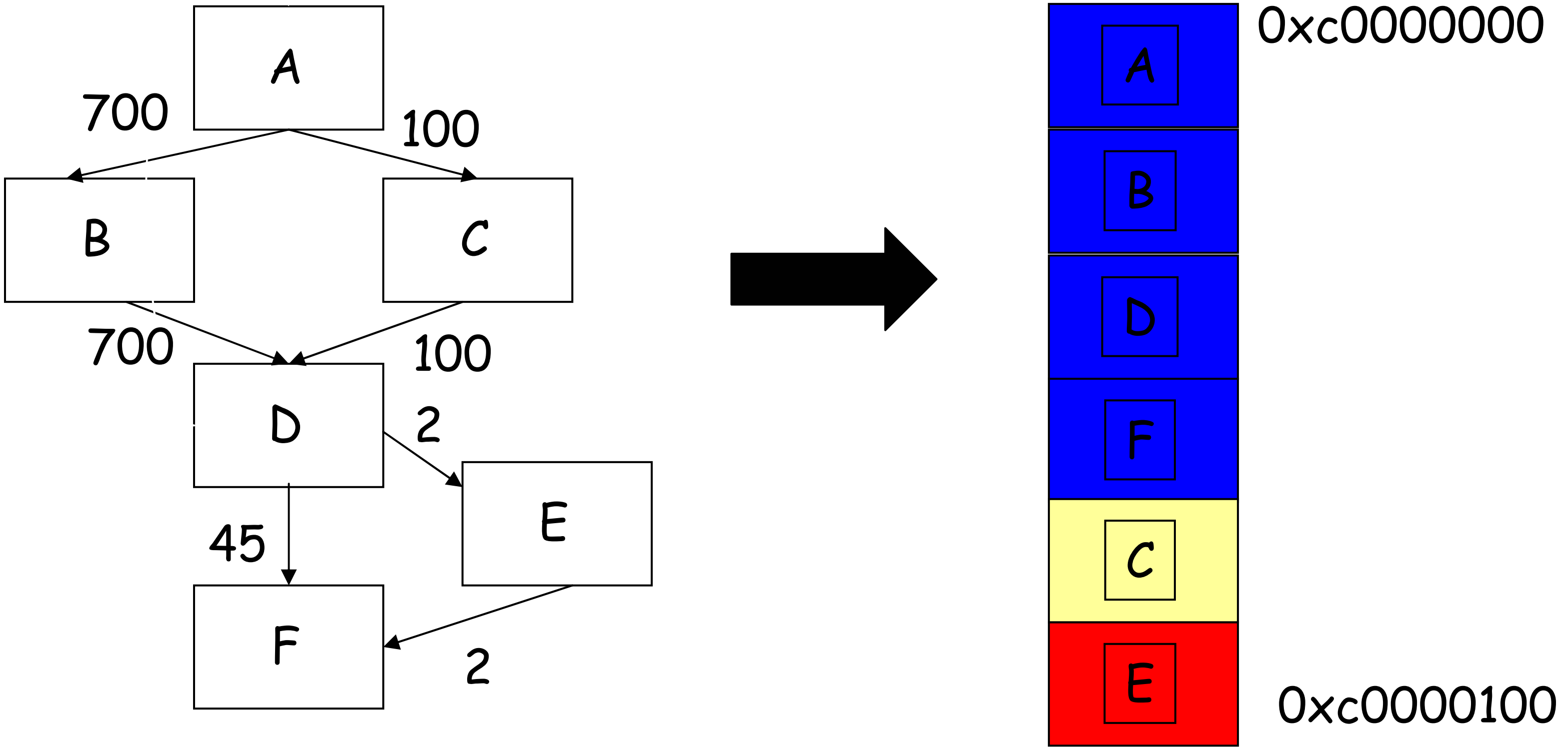
- May be desirable to customize inlining heuristic based on data flow effect

Solution: "Empirical" optimization

- Compiler tentatively inlines a call site
- Subsequently monitors compiler transformations to quantify effect on optimization
- Future inlining decisions based on past effects

Code positioning

- Archetype: Pettis and Hansen [PLDI 90]
- Easy and profitable: employed in most (all?) production VMs
- Synergy with trace scheduling [eg. Star-JIT/ORP]

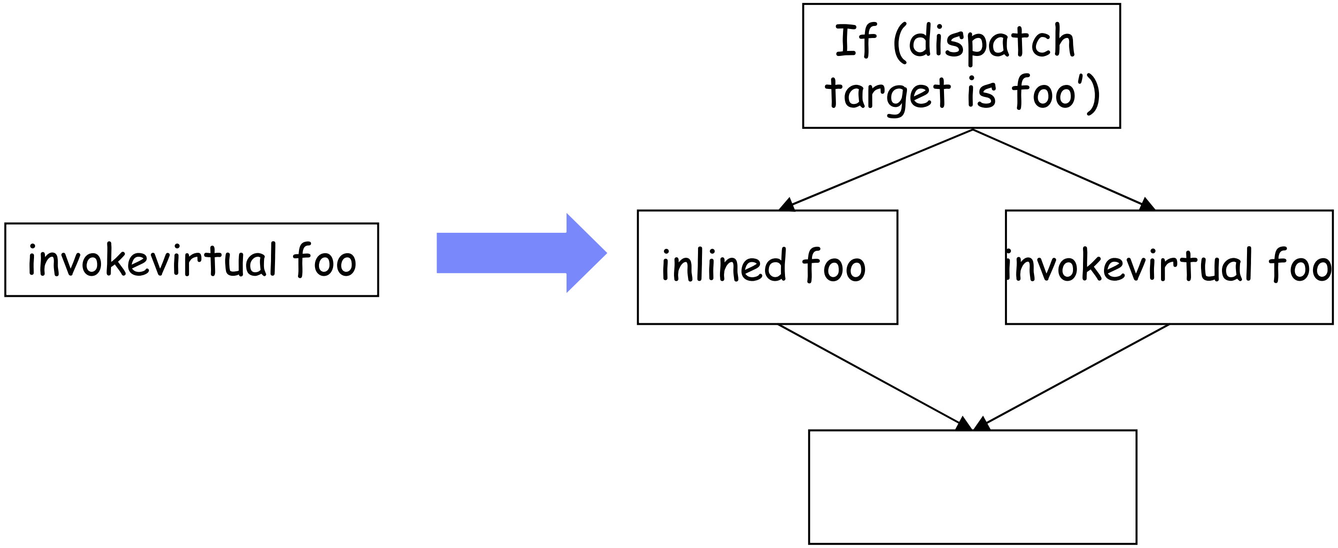


Multiversioning

- Compiler generates multiple implementations of a code sequence
  - Emits code to choose best implementation at runtime
- Static Multiversioning
  - All possible implementations generated beforehand
  - Can be done by static compiler
  - FDO: Often driven by profile-data
- Dynamic Multiversioning
  - Multiple implementations generated on-the-fly
  - Requires runtime code generation

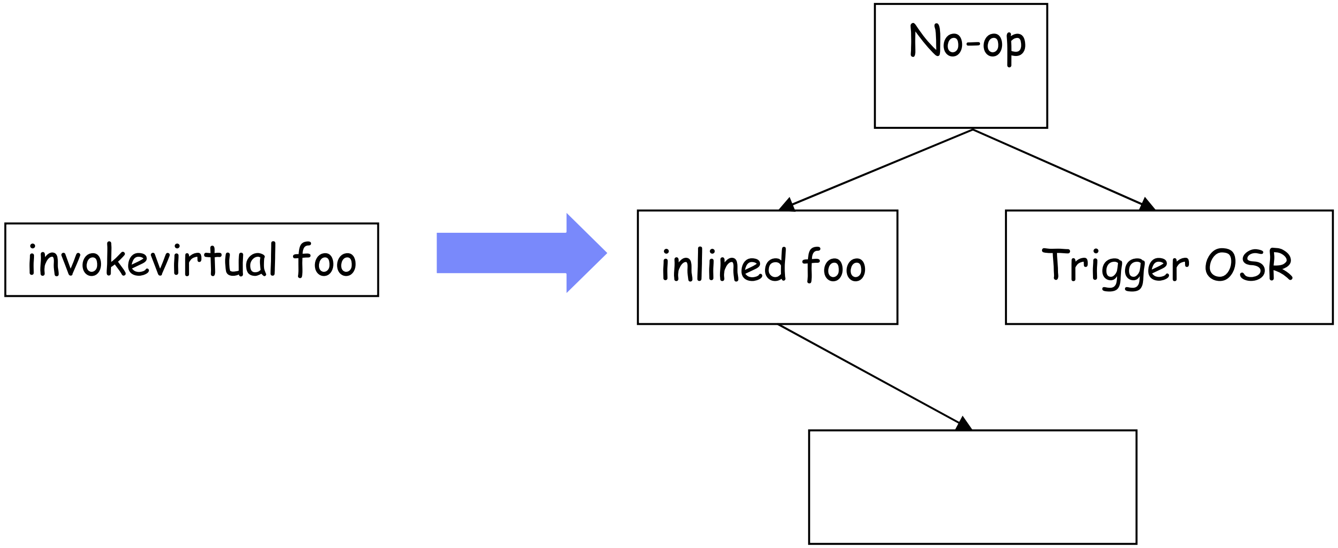
Static Multiversioning Example

- Guarded inlining for a virtual method w/ dynamic test
- Profile data indicates mostly monomorphic call sites
- Note that downstream merge pollutes forward dataflow



Static Multiversioning with On-Stack Replacement [SELF, HotSpot, Jikes RVM]

- Guarded inlining for a virtual method w/ patch point & OSR
  - Patch no-op when class hierarchy changes
  - Generate recovery code at runtime (more later)
- No downstream merge -> better forward dataflow



Dynamic Multiversioning: Customization in SELF

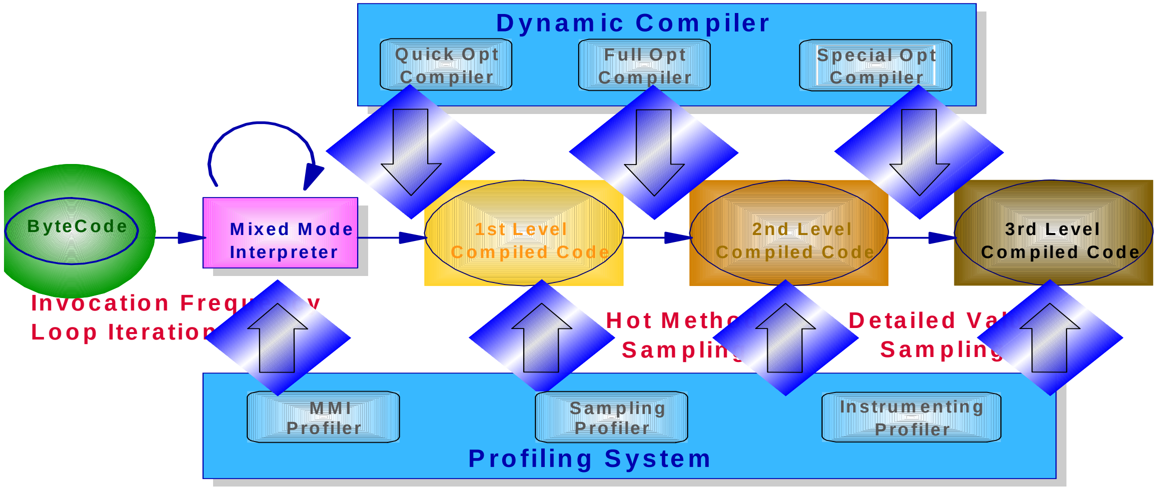
- Generate new compiled version of a method for each possible receiver class on first invocation with that receiver

- Mostly targeted to eliminating virtual dispatch overhead
  - Know precise type for 'self' (this) when compiling

- Works well for small programs, scalability problems
  - Naïve approach eventually abandoned
  - Selective profile-guided algorithm later developed in Vortex [Dean et al. '95]

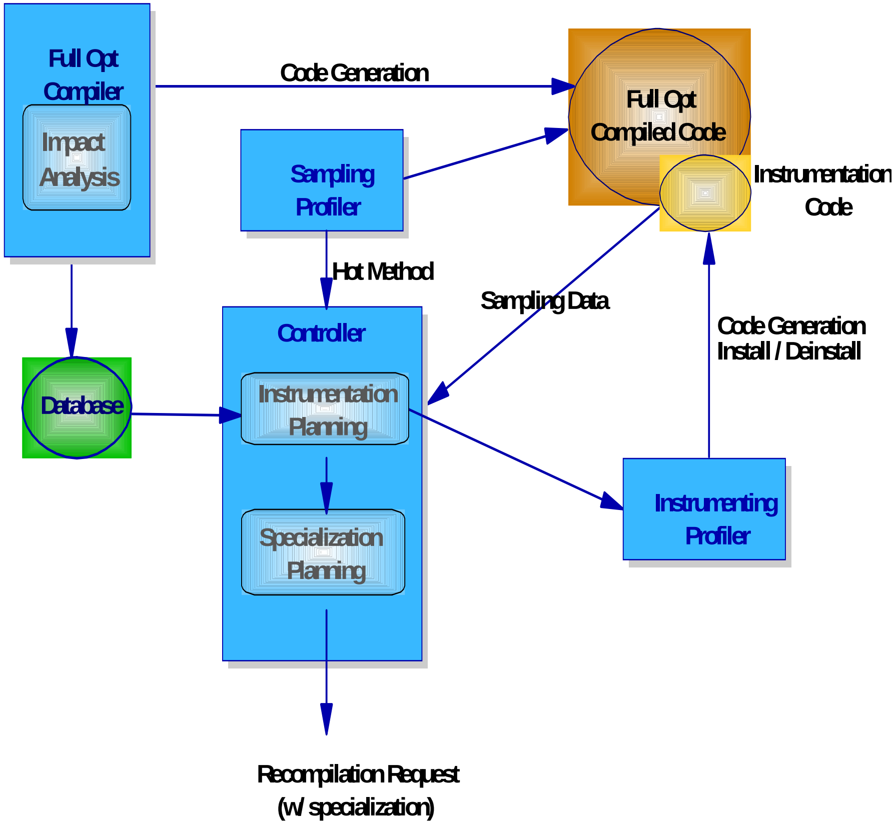
IBM DK for Java with FDO [Suganuma et al. '01]

- MMI (Mixed Mode Interpreter)
  - Fast interpreter implemented in assembler
- Quick compilation
  - Reduced set of optimizations
- Full compilation
  - Full optimizations for selected hot methods
- **Special compilation**
  - ***Code specialization based on value profiling***



Specialization: IBM DK [Suganuma et al. '01]

- For hot methods, compiler performs **"impact analysis"** to evaluate potential specializations
    - Parameters and statics
- For desirable specializations, compiler dynamically installs instrumentation for value profiling
- Based on value profile, compiler estimates if specialization is profitable and generates specialized versions
- Process can iterate



Impact Analysis

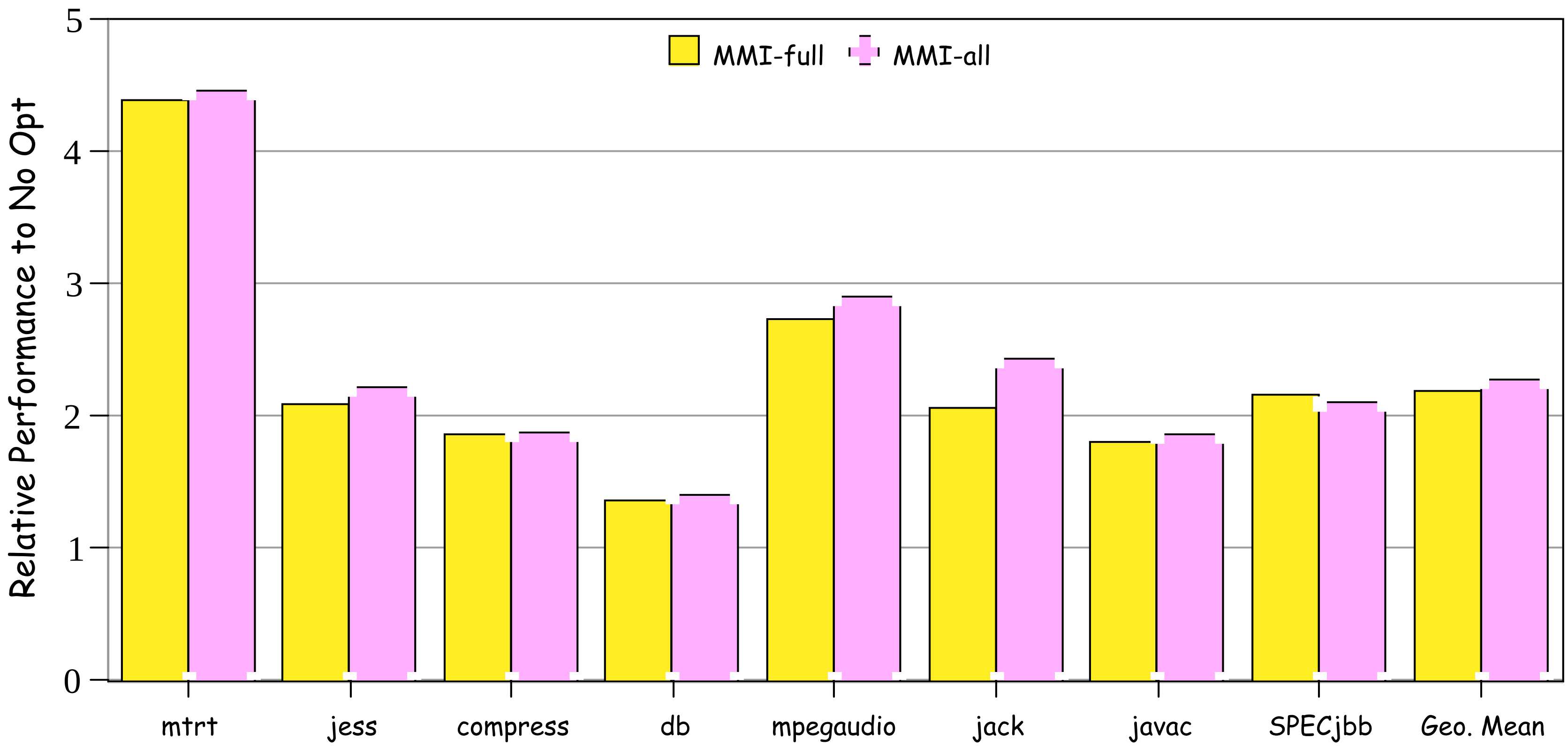
- **Problem:** When is specialization profitable?

- **Impact analysis:** Compute estimate of code quality improvement if we knew a specific value or type for some variables
  - Constant Value of Primitive Type
    - Constant Folding, Strength Reduction (div, fp transcendental)
    - Elimination of Conditional Branches, Switch Statements
  - Exact Object Type
    - Removal of Unnecessary Type Checking Operations
    - CHA Precision Improvement -> Inlining Opportunity
  - Length of Array Object
    - Elimination or Simplification of Bound Check Operations
    - Loop Simplification

- Dataflow algorithm

- For each possible specialization target (variable), compute how many statements could be eliminated or simplified

Steady State: IBM DK for Java + FDO/Specialization [Suganuma et al.'01]



FDO Potpourri

Many opportunities to use profile info during various compiler phases
Almost any heuristic-based decision can be informed by profile data

Examples:

- Loop unrolling
  - Unroll “hot” loops only
- Register allocation
  - Spill in “cold” paths first
- Global code motion
  - Move computation from hot to cold blocks
- Exception handling optimizations
  - Avoid expensive runtime handlers for frequent exceptional flow
- Speculative stack allocation
  - Stack allocate objects that escape only on cold paths
- Software prefetching
  - Profile data guides placement of prefetch instructions

Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
    - Gathering profile information
    - Exploiting profile information in a JIT
        - Feedback-directed optimizations
        - **Aggressive speculation and invalidation**
    - Exploiting profile information in a VM
5. Summing Up and Looking Forward

Example: Class hierarchy based inlining

```
longRunningMethod ( ) {
        Foo foo = getSomeObject();
        foo.bar();
}
```

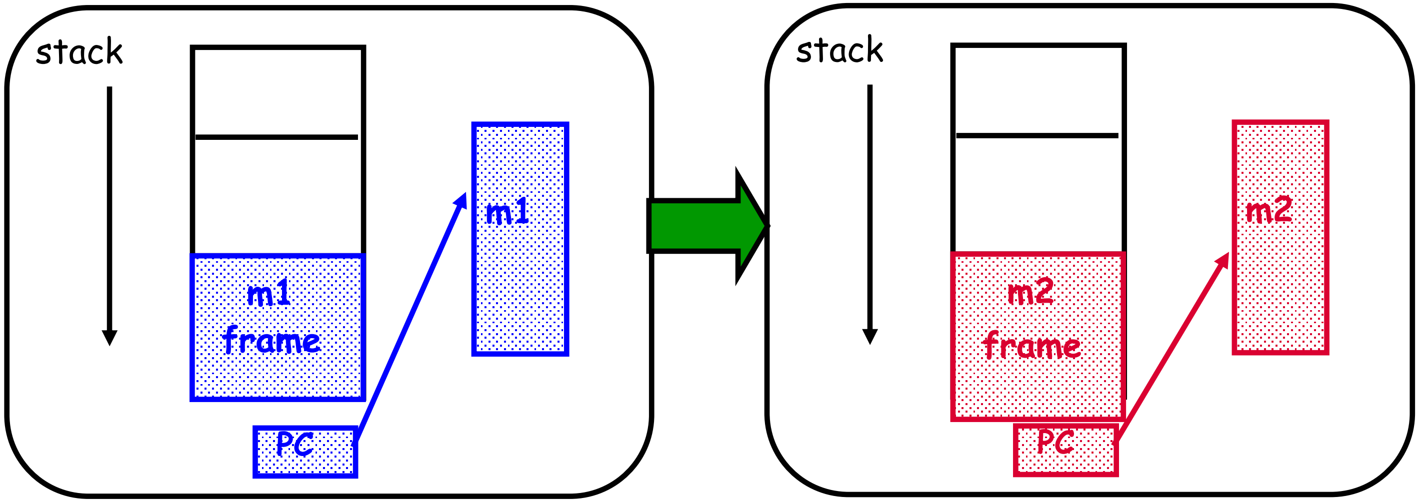
- According to current class hierarchy
  - Only one possible virtual target for **foo.bar()**
  - Idea: speculate that class loading won't occur
    - Inline Foo::bar()
  - Monitor class loading: if Foo::bar() is overridden
    - Recompile all methods containing incorrect code
  - But what if longRunningMethod never exits?
    - One option: *on-stack replacement*

Invalidation via On-Stack Replacement (OSR)

[Chambers,Hölzle&Ungar'91-94, Fink&Qian'03]

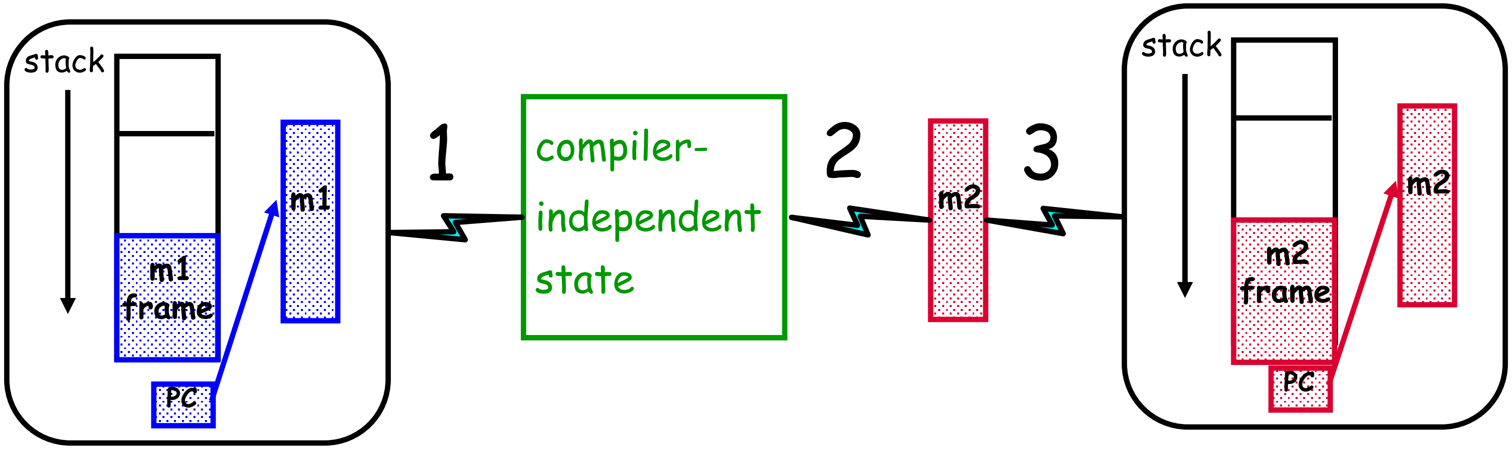
Transfer execution from compiled code **m1** to compiled code **m2** even while **m1** runs on some thread's stack

Extremely general mechanism ➔ minimal restrictions on speculation



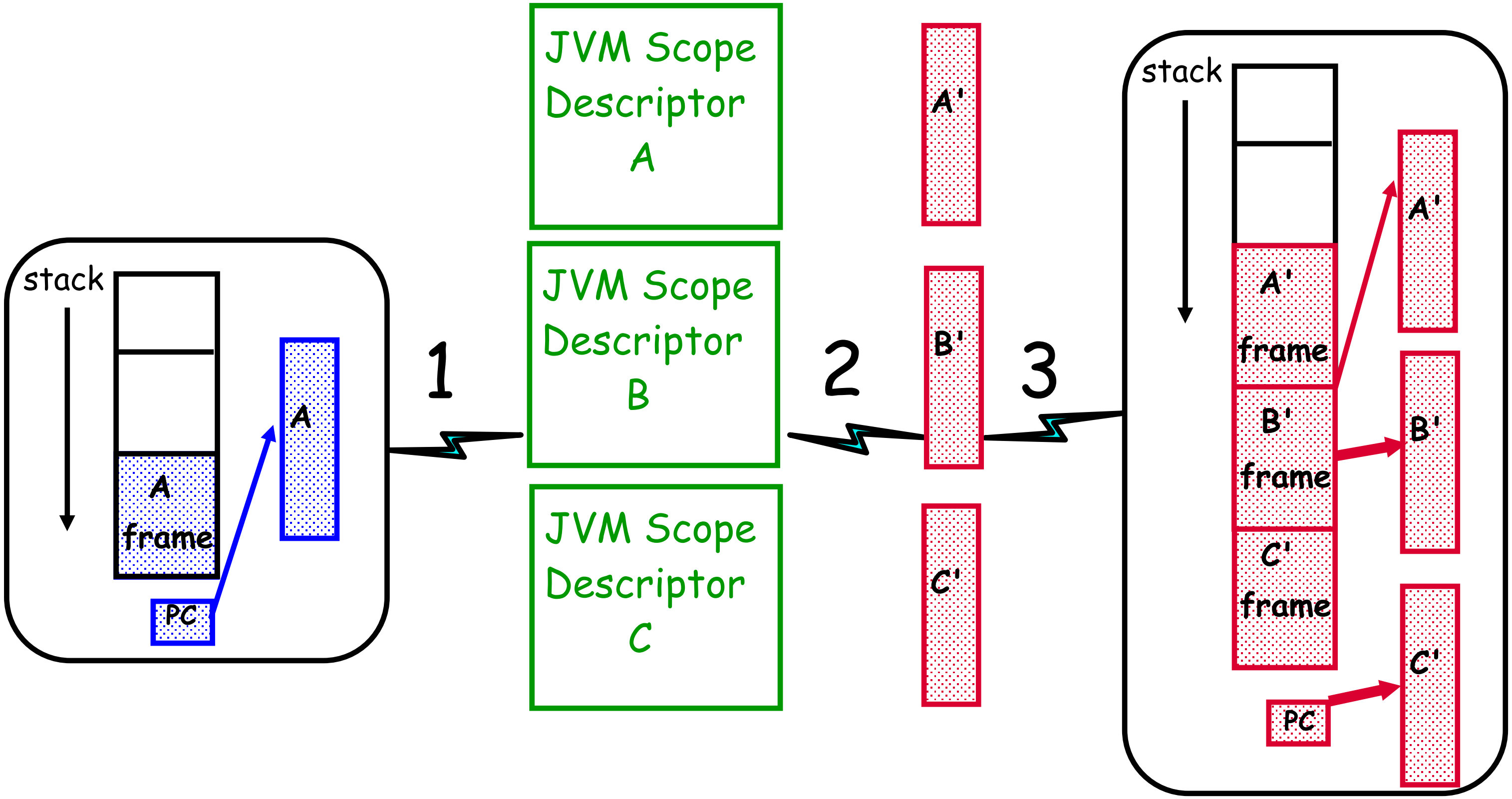
OSR Mechanisms

- Extract compiler-independent state from a suspended activation for m1
- Generate new code m2 for the suspended activation
- Transfer execution to the new code m2



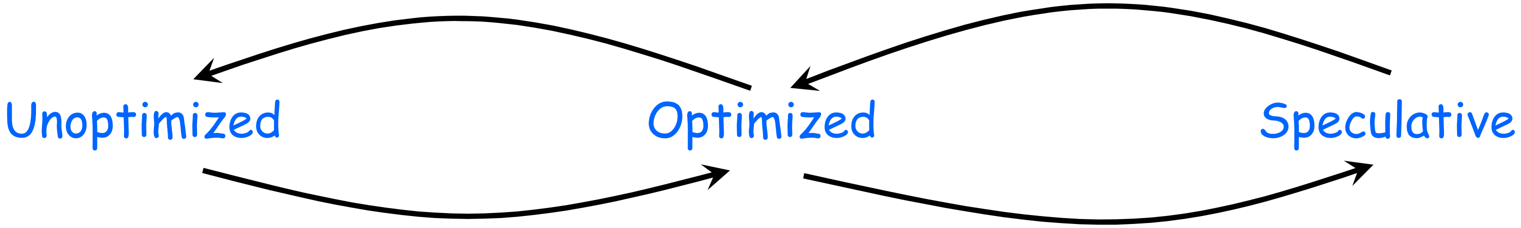
OSR and Inlining

Suppose optimizer inlines A → B → C:



Applications of OSR

1. Safe invalidation for speculative optimization
    - Class-hierarchy-based inlining [HotSpot]
    - Deferred compilation [SELF-91, HotSpot, Whaley 2001]
        - Don't compile uncommon cases
        - Improve dataflow optimization and reduce compile-time
2. Debug optimized code via dynamic deoptimization [Holzle et al. '92]
    - At breakpoint, deoptimize activation to recover program state
3. Runtime optimization of long-running activations [SELF-93]
    - Promote long-running loops to higher optimization level



Unoptimized Optimized Speculative

Invalidation Discussion

- OSR challenges
  - Nontrivial to engineer
  - Code that is both complex and infrequently executed is a prime location for bugs
  - Keeping around extra state may introduce overhead
- Other existing invalidation techniques
  - Pre-existence inlining [Detlefs&Agesen'99]
  - Code patching [Suganama'02]
  - Thin Guards [Arnold&Ryder'02]
- Once invalidation mechanism exists
  - Relatively easy to perform speculative optimizations
  - Many researchers avoid interprocedural analysis of Java for the wrong reasons
    - Invalidation is "easy". The fun parts are
      - Must be able to detect when assumptions change
      - Must be low overhead, incremental
    - Area mostly unexplored (Hirzel et al.,'04)

Invalidation via pre-existence [Detlefs & Agesen'99]

- When applicable, enables all of the benefits of OSR, without the complexities of a full OSR implementation.

```
int foo(A a) {
        ......
        a.m1();
}
```

- Key insight: if inlining m1 without a runtime guard is valid when foo is invoked, it will be valid when the inlined code executes
  - Exploiting "pre-existence" of object reference by a
- Invalidation is required only for all future invocations
  - No interrupted activations a la OSR

Dynamic Class Hierarchy Mutation [Su and Lipasti, 06]

- Idea:
  - Find methods with control flow dependent on some "state" field
  - Create specialized methods for the different values
  - Use virtual function dispatch
- Implementation
  - Offline
    - Finds hot methods with control dependent on states whose value is set in cold methods
    - Capture values and distribution of states (using sampling)
  - Online
    - JVM specializes hot methods with hot values by dispatching to the specialized method at runtime
      - Tracks assignments of hot fields (for opportunities and invalidation)
      - Modifies virtual function table to specialized implementation
  - Incorporated into an existing adaptive optimization system

Dynamic Class Hierarchy Mutation [Su and Lipasti, '06]

Results
- Benchmarks: SPECjbb2000, SPECjbb2005, 4 other programs
- 2 to ~8% performance improvement
  - author-created benchmark shows over 30% improvement
- ~1.5—7% code size increase
- ~2-17% compilation time increase

Assessment
- Interesting idea
- Specialization regions are limited to methods (uses virtual dispatch), but system creates these methods
- How do you do this online?

Runtime Specialization With Optimistic Heap Analysis

[Shankar et al., OOPSLA'05]

Online technique, first to track heap variables
Motivation: specialization of "interpreter" programs

Algorithm

1. Find a specialization starting point in a hot function

2. Specialize: create a trace for each hot value k
   - Loops unrolled, branch prediction for nonconstant conditionals
   - Eliminate loads from invariant memory locations
   - Eliminates safety checks, dynamic dispatch, etc.
   - Modify dispatch to select appropriate trace

3. Invalidate when assumed invariant locations are updated

Finding Specialization Points

- The best point can be near the end of the function
- Ideally: try to specialize from all instructions
  - Pick the best one, as defined by "Influence"
  - Influence(e) = Expected number of dynamic instructions from the first occurrence of epc to the end of the function
    - Dataflow-independent
- System of equations, solved in linear time

Finding Invariant Memory Locations

- Provides the bulk of the speedup
- Existing work relied on static analysis or annotations
- Solution: sampled invariance profiling
  - Track every nth store
  - Locations detected as written: not constant
  - Everything else: **optimistically** assumed constant
- 95.6% of claimed constants remained constant

- Use Arnold-Ryder duplication-based sampling to gather other useful info
  - CFG edge execution frequencies
    - Helps identify good trace start points (influence)
  - Hot values at particular program points
    - Helps seed the constant propagator with initial values

Invalidation

- Because heap analysis is optimistic
  - Need to guard assumed constant locations
  - And invalidate corresponding traces
- Solution to the two key problems
  - Detect when such a location is updated
    - Use write barriers (type information eliminates most barriers)
    - Overhead: 0-12%
  - Invalidate corresponding specialized traces
    - A bit tricky: trace may need to be invalidated while executing
    - Uses OSR

Results

Benchmark	**Input**	**Speedup**
`convolve` Transforms an image with a matrix; from the ImageJ toolkit	fixed image, various matrices	**2.74x**
	fixed matrix, various images	**1.23x**
`dotproduct` Converted from C version in DyC	sparse constant vector	**5.17x**
`interpreter` Interprets simple bytecodes	bubblesort bytecodes	**5.96x**
	binary search bytecodes	**6.44x**
`jscheme` Interprets Scheme code	partial evaluator	**1.82x**
`query` Performs a database query; from DyC	semi-invariant query	**1.71x**
`sim8085` Intel 8085 Microprocessor simulator	included sample program	**1.70x**
`em3d` (intentionally unspecializable) Electromagnetic wave propagation	-n 10000 -d 100	**0.98x**

Runtime Specialization With Optimistic Heap Analysis
[Shankar et al., OOPSLA'05]

Assessment

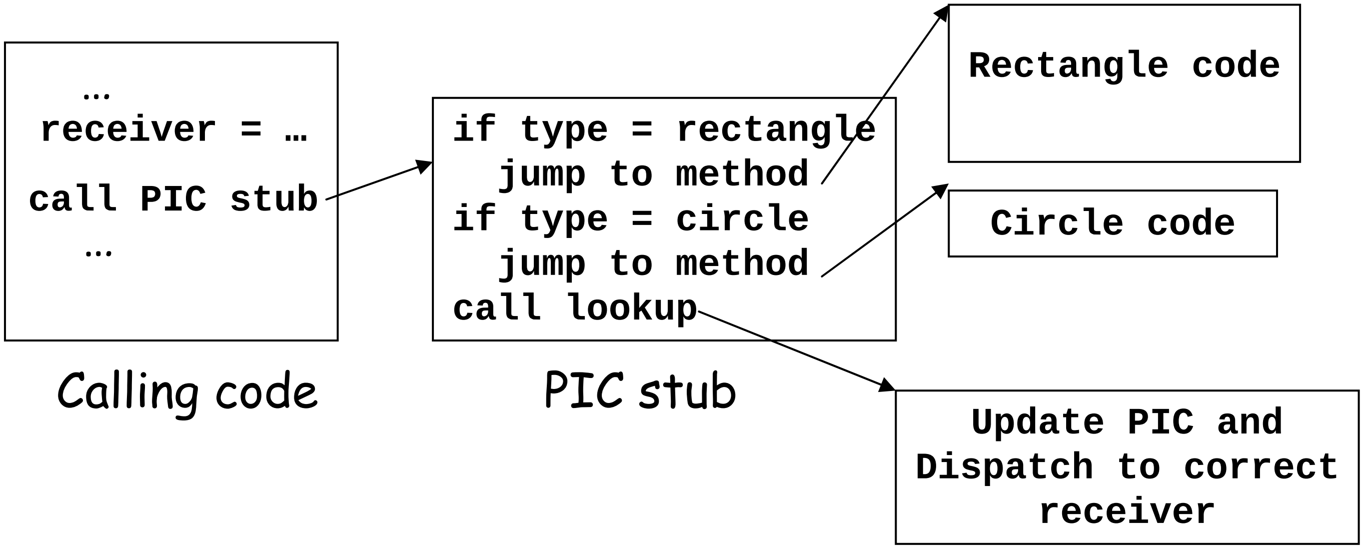
- Completely online, usable in a JVM
- More optimistic approach
- Effective on interpreter programs
  - What about general commercial applications?
  - Need to overcome overhead
- Current state of the art in online specialization

Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
   - Gathering profile information
   - Exploiting profile information in a JIT
     - Feedback-directed optimizations
     - Aggressive speculation and invalidation
   - **Exploiting profile information in a VM**
     - Dispatch optimizations
     - Speculative object models
     - GC and locality optimizations
5. Summing Up and Looking Forward

Virtual/Interface Dispatch

- Polymorphic inline cache [Holzle et al.'91]



```
…
receiver = …
call PIC stub
…
```

Calling code

```
if type = rectangle
   jump to method
if type = circle
   jump to method
call lookup
```

PIC stub

Rectangle code

Circle code

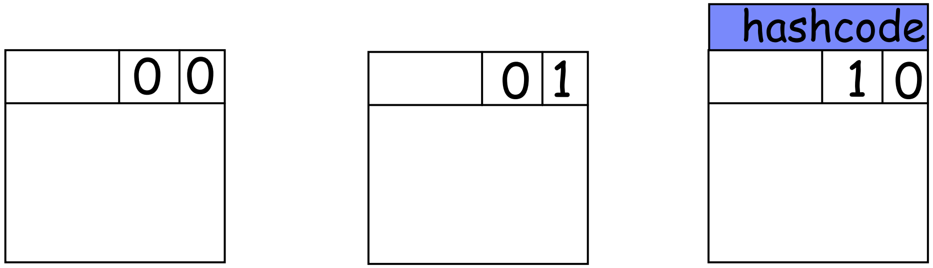
Update PIC and Dispatch to correct receiver

Requires limited dynamic code generation

Speculative Meta-data Representations

Example: Object models

- Tri-state hash code encoding [Bacon et al. '98, Agesen Sun EVM]

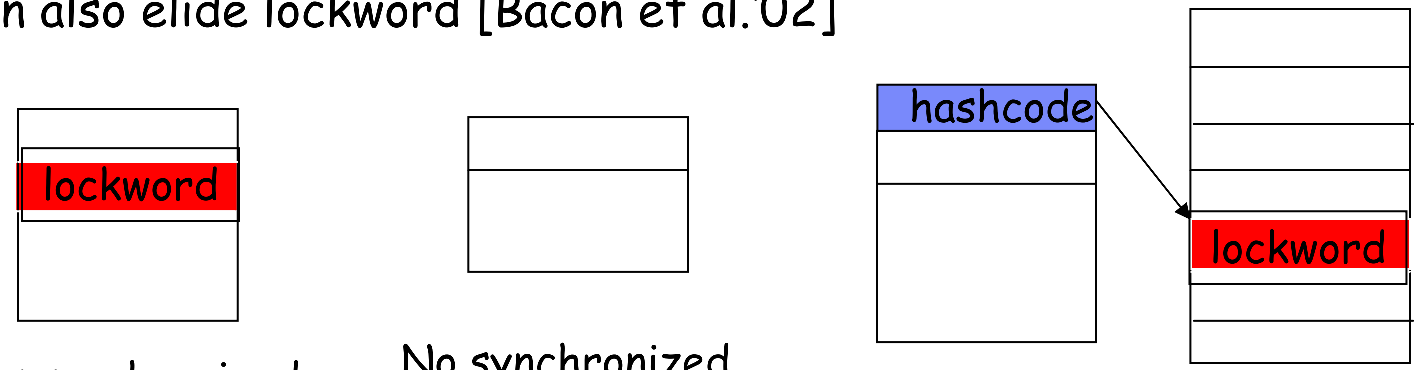


Unhashed

Hashed
(hashcode == address)

Hashed
and Moved

- Can also elide lockword [Bacon et al.'02]



Has synchronized
method

No synchronized
method

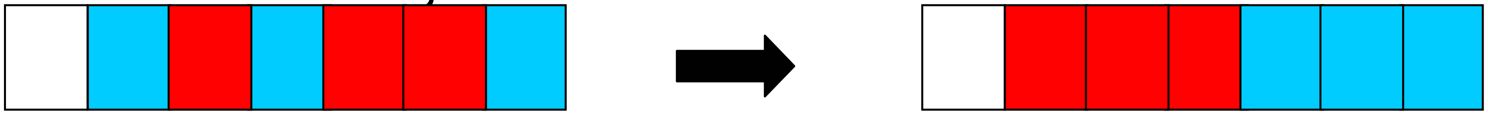
No synchronized
method, but locked

Adaptive GC techniques

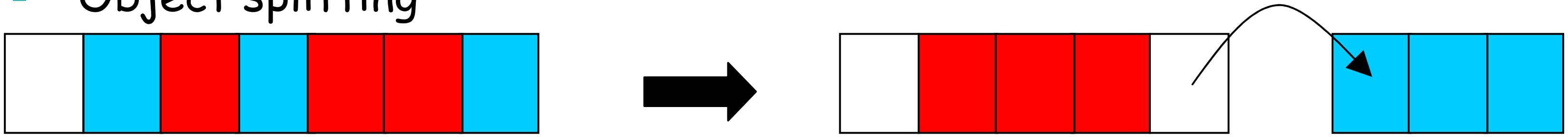
- Dynamically adjust heap size
  - IBM DK [Dimpsey et al. ‘00] – policy depends on heap utilization and fraction of time spent in GC
- Switch GC algorithms to adjust to application behavior
  - [Printezis ‘01] – switch between Mark&Sweep and Mark&Compact for mature space in generational collector
  - [Soman et al.’03] – more radical approach prototyped in Jikes RVM
  - Not yet exploited in production VMs
- Opportunistic GC
  - [Hayes’91] – key objects keep large data structures live
  - Not yet exploited in production VMs

Spatial Locality Optimizations

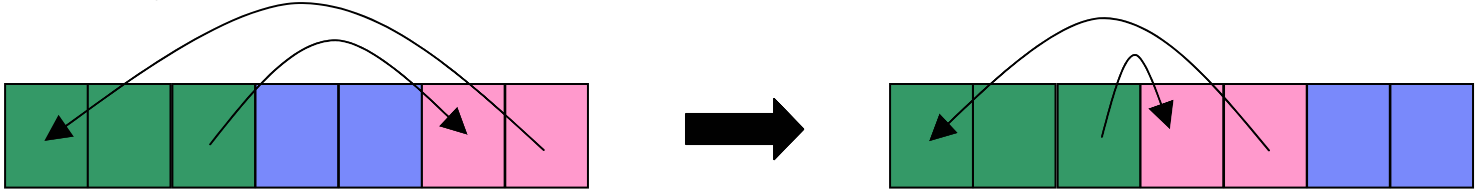
- Move objects, change objects to increase locality, or prefetch
- Field reordering



- Object splitting



- Object co-location



Spatial Locality Optimizations

- Examples
  - Kistler & Franz '00
  - Chilimbi et al., '99
  - Huang et al. '04
  - Adl-Tabatabai et al. '04
  - Chilimbi & Shahan '06
  - Siegwart & Hirzel '06
  - Etc.

- Very hot area
- Encouraging results, some with offline profiling, some online
- Example of getting hardware and VM to work better together

Course Outline

1. Background
2. Engineering a JIT Compiler
3. Adaptive Optimization
4. Feedback-Directed and Speculative Optimizations
5. **Summing Up and Looking Forward**
    - **Debunking myths**
    - The three waves of adaptive optimization
    - Future directions

Debunked Myths

1. Because they execute at runtime, dynamic compilers must be blazingly fast
2. Dynamic class loading is a fundamental roadblock to cross-method optimization
3. Sophisticated profiling is too expensive to perform online
4. A static compiler will always produce better code than a dynamic compiler
5. Infrastructure requirements stifle innovation in this field
6. Production VMs avoid complex optimizations, favoring stability over performance

Myths Revisited I

Myth: Because they execute at runtime dynamic compilers must be blazingly fast.

- they cannot perform sophisticated optimizations, such as SSA, graph-coloring register allocation, etc.

Reality:

- Production JITs perform all the classical optimizations
- Language-specific JITs exploit type information not available to C compilers (or 'classic' multi-language backend optimizers)
- Selective optimization strategies successfully focus compilation effort where needed

Myths Revisited II

Myth: Dynamic class loading is a fundamental roadblock to cross-method optimization:

- Because you never have the whole program, you cannot perform interprocedural optimizations such as virtual method resolution, virtual inlining, escape analysis

Reality:

- Can speculatively optimize with respect to current class hierarchy
- Sophisticated invalidation technology well-understood; mitigates need for overly conservative assumptions
- Speculative optimization can be more aggressive than conservative, static compilation

Myths Revisited III

Myth: Sophisticated profiling is too expensive to perform online

Reality:

- Sampling-based profiling is cheap and can collect sophisticated information
- e.g. Arnold-Ryder full-duplication framework
- e.g. IBM DK dynamic instrumentation

Myths Revisited IV

Myth: A static compiler can always get better performance than a dynamic compiler because it can use an unlimited amount of analysis time.

Reality:

- Production JITs can implement all the classical optimizations static compilers do
- Feedback-directed optimization should be more effective than unlimited IPA without profile information
- Legacy C compiler backends can't exploit type information and other semantics that JITs routinely optimize
- However, ahead-of-time compilation still needed sometimes:
  - Fast startup of large interactive apps
  - Small footprint (e.g. embedded) devices
- Incorporating ahead-of-time compilation into full-fledged VM is well-understood

Myths Revisited V

Myth: Small independent academic research group cannot afford infrastructure investment to innovate in this field

Reality:

- High-quality *open-source* virtual machines are available
  - Jikes RVM, ORP, Kaffe, Mono, etc.
  - Apache Harmony looks interesting

Myth VI - Production VMs avoid complex optimizations, favoring stability over performance

Perception: Complex, speculative optimizations introduce hard to find bugs and are not worth the marginal performance returns.

Reality: There is pressure to obtain high performance

- Production JVMs perform many complex optimizations, including
  - Optimizations that require sophisticated coding
  - Difficult to debug dynamic behavior
    - e.g., nondeterministic profile-guided optimizations
  - Speculative optimizations involving runtime invalidation
- Production JVM's are leading the field in VM performance
  - Often ahead of academic and industrial research labs

This does not mean there are no problems

- Commercial VMs do dynamic, cutting-edge optimizations, but..
  - Complexity of VMs keeps growing
    - Layer upon layer of optimizations with potential unknown interactions
  - Often:
    - Solutions may not be the most general or robust
      - Targeted to observed performance problems
    - Not evaluated with the usual scientific rigor
      - Not published
    - See performance “surprises” on new applications

- There are many research issues that academic researchers could help explore:
  - Performance, robustness, and stability
    - Would really help the commercial folks

How much performance gain is interesting?

- Quiz: An optimization needs to produce > X% performance improvement to be considered interesting.   X = ?
  - a) 1% b) 5% c) 10% d) 20%
  - Sometimes research papers with < 5-10% improvement are labeled failures
- Answer: it depends on complexity of the solution
  - Value = performance gain / complexity
  - Every line of code requires maintenance, and is a possible bug
    - 10 LOC yielding 1.5% speedup
      - Product team may incorporate in VM by end of week
    - 25,000 LOC yielding 1.5% speedup:
      - Not worth the complexity
- Improving performance with reduced complexity is important
  - Needs to be rewarded by program committees

Comparison Between HLL VMs and Dynamic Binary Optimizers

HLL VM	Dynamic Binary Optimizer
Applies to programs in target languages	Applies to any program
Exploits program structure and high-level semantics (e.g. types)	Views stream of executed instructions, can infer limited program structure and low-level semantics
Large gains from runtime optimization (10X vs. interpreter)	Smaller gains from runtime optimization (10% would be good?)
Most effective optimizations: inlining, register allocation	Most effective optimizations: instruction scheduling, code placement
Optimizer usually expensive, employed selectively	Optimizer usually cheap, often employed ubiquitously

Trends suggest that more programs will be written to managed HLLs
  - For such programs, does binary optimizer add value?

Chen et al [CGO'06] combine both

Waves of Adaptive Optimization

1. Use JIT to compile all methods (Smalltalk-80)

2. Selective Optimization (Adaptive Fortran, Self-93)
   - Use many JIT levels to tradeoff cost/benefits of various optimizations
   - Exploit 80-20 rule
   - *limits the **costs** of runtime compilation*

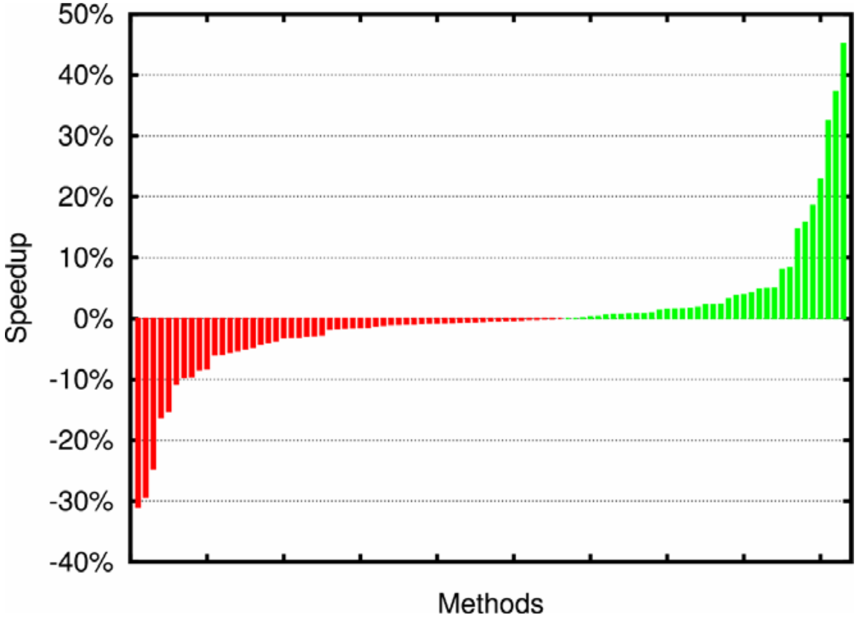
3. Online FDO (Today's JVMs)
   - Use profile information of **current** run to improve optimization accuracy
   - *exploits **benefit** of runtime compilation*

4. What is the next wave?

The 4th Wave of Adaptive Optimization?

- Try multiple optimization strategies for a code region, **online**
- Run and time all versions online
- Determine which performs the best
- Use it in the future
- Examples
  - Dynamic Feedback [Diniz & Rinard, '97]
    - Measure synchronization overhead of each version
  - ADAPT [Voss & Eigenmann '01]
    - Uses fastest executed version after partitioning timings into bins
  - Fursin et al. '05
    - Measure two versions after a stable period of execution is entered
  - Performance Auditor [Lau et al. '06]
    - *More details to follow*

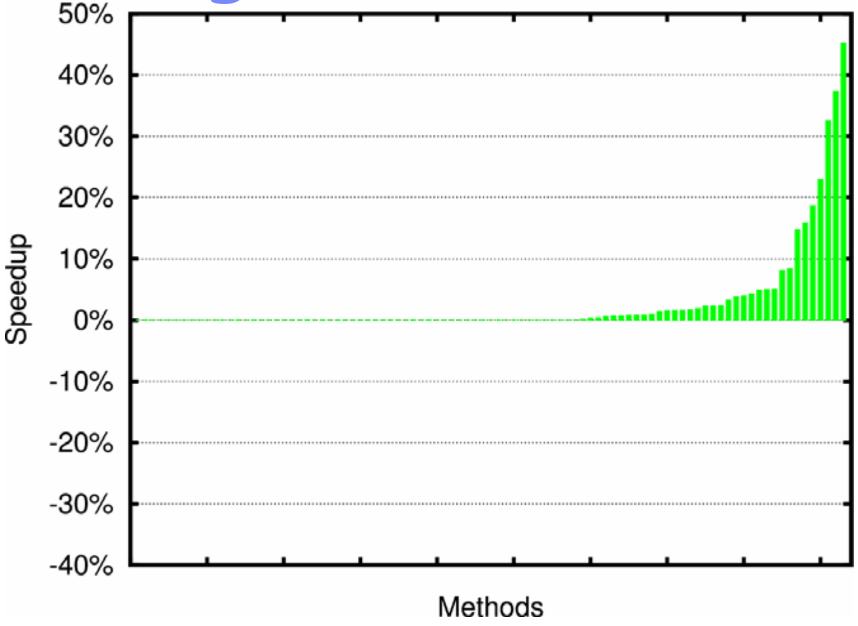
Performance Auditor



Per-Method speedups
Aggressive inlining vs. default inlining (J9 JVM, 100 hot methods)

- Aggressive inlining: mixed results
- More slowdowns than speedups
- But not a total loss – there are significant speedups!

Wishful Thinking

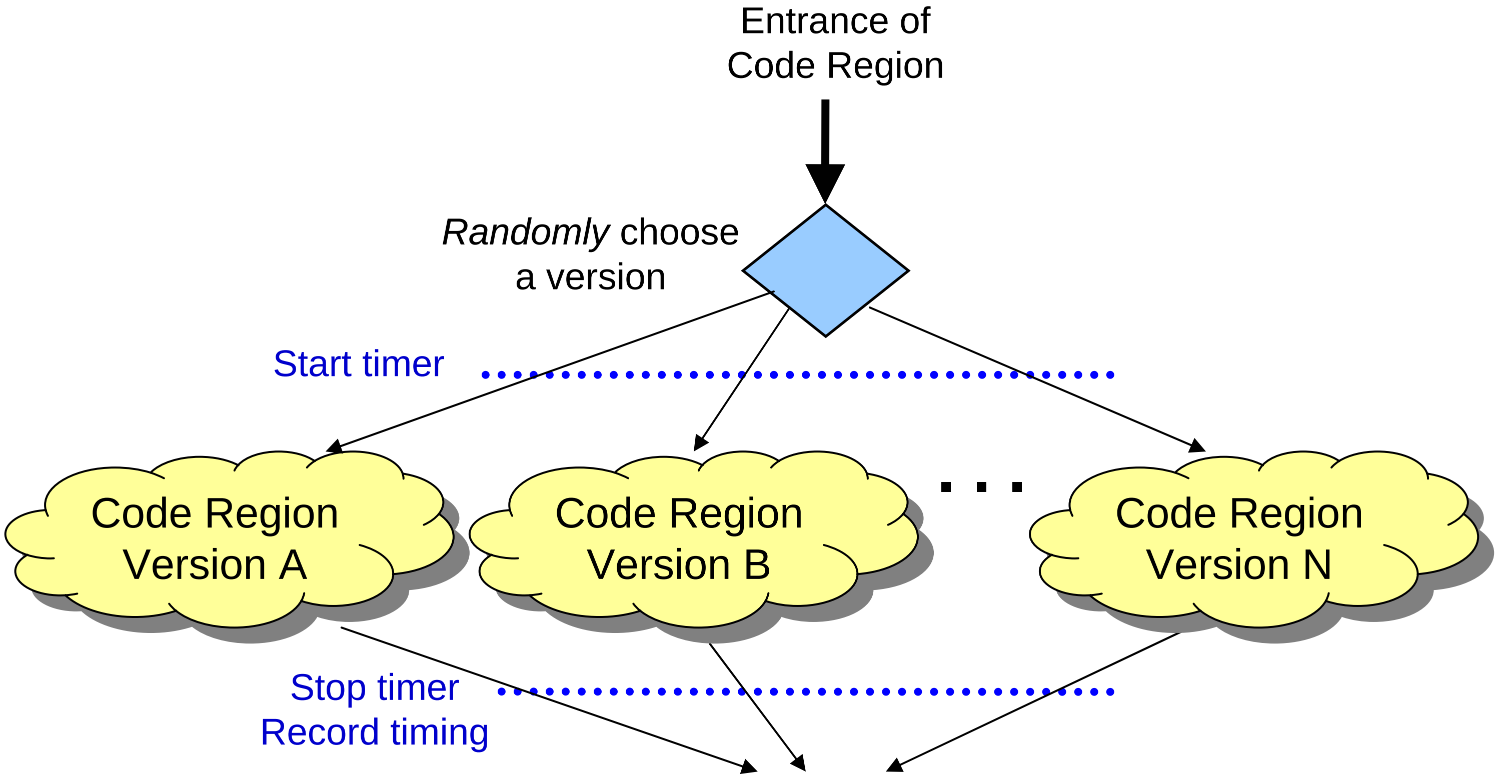


- Dream: A world without slowdowns
- Default inlining heuristics miss these opportunities to improve performance
- Goal: Be aggressive only when it produces speedup

Challenge

- Which implementation is fastest?
  - Decide online, without stopping and restarting the program
- Can't just invoke each version once and compare times
  - Changing inputs, global state, etc
- Example: Sorting routine. Size of input determines run time
  - SortVersionA(10 entries) vs SortVersionB(1,000,000 entries)
  - Invocation timings don't reflect performance of A and B
    - Unless we know that input size correlates with runtime
    - But that requires high-level understanding of program behavior
- Solution: Collect multiple timing samples for each version
  - Use statistics to determine how many samples to collect

Timing Infrastructure Design



Entrance of
Code Region

Randomly choose
a version

Start timer

Code Region
Version A

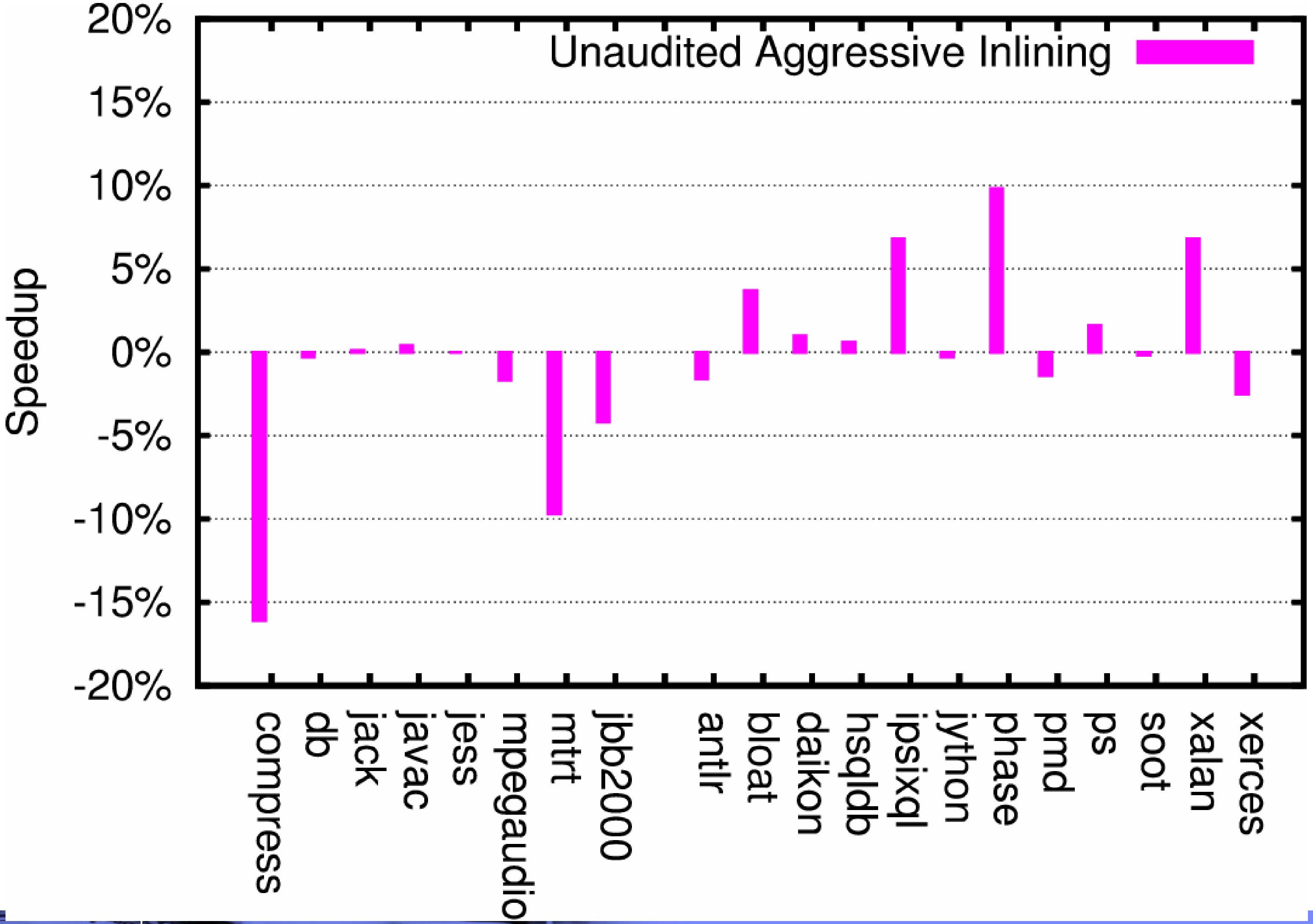
Code Region
Version B

. . .

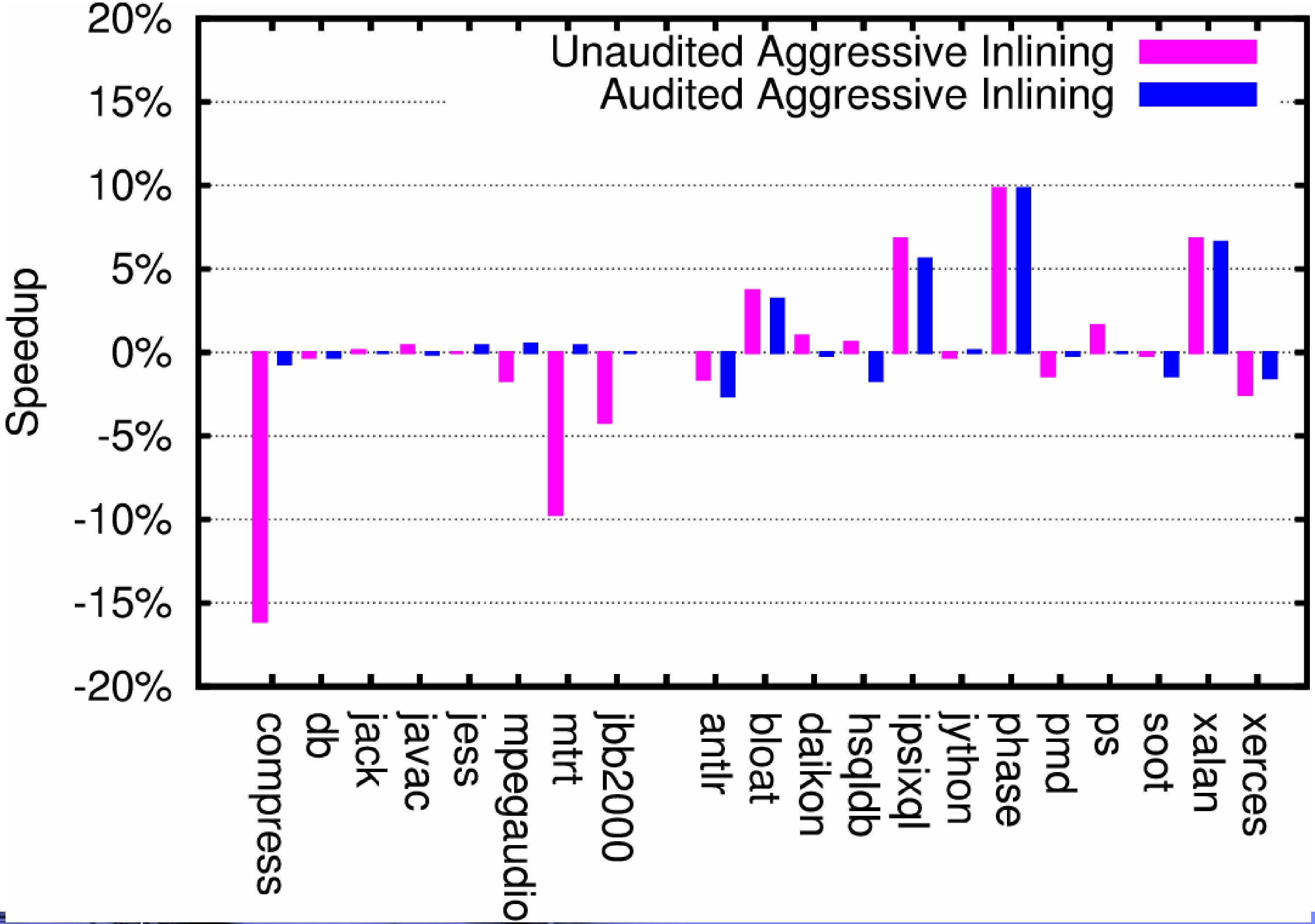
Code Region
Version N

Stop timer
Record timing

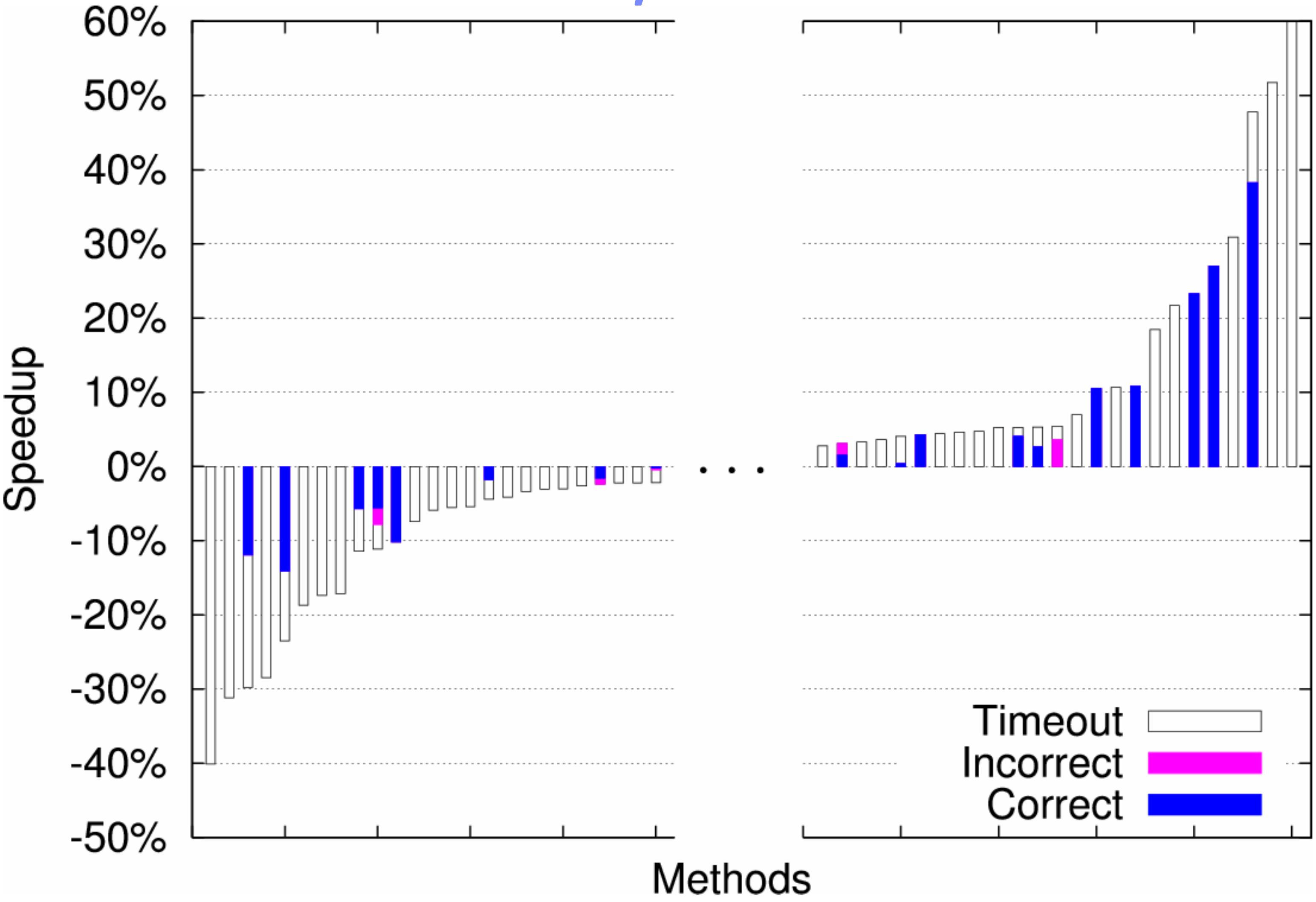
Results


Unaudited Aggressive Inlining
Speedup
20%
15%
10%
5%
0%
-5%
-10%
-15%
-20%
compress
db
jack
javac
jess
mpegaudio
mtrt
jbb2000
antlr
bloat
daikon
hsqldb
ipsixql
jython
phase
pmd
ps
soot
xalan
xerces

Results


Unaudited Aggressive Inlining
Audited Aggressive Inlining
Speedup
20%
15%
10%
5%
0%
-5%
-10%
-15%
-20%
compress
db
jack
javac
jess
mpegaudio
mtrt
jbb2000
antlr
bloat
daikon
hsqldb
ipsixql
jython
phase
pmd
ps
soot
xalan
xerces

Per-Method Accuracy



No shortage of research problems for virtual machines (1/2)

- Higher-level optimizations
  - General purpose components, using tiny fraction of functionality
  - Higher-level programming models (e.g. J2EE, XML, Web Services, BPEL)
- Traditional optimizations, but for non-"toy" benchmarks
  - Selective optimization for programs with 30,000 methods
  - Inlining for call stack > 200 deep
- More aggressive use of speculation
  - Dynamic compiler looks too much like traditional static compilers
- Stability of performance
  - Too many ad-hoc optimizations based on (poorly tuned) heuristics
  - React to phase shifts

No shortage of research problems for virtual machines (2/2)

- Optimizations for locality
  - New challenges and opportunities in managed runtimes
- Online interprocedural analysis
  - Mostly unexplored
  - Take a more global view of optimization
- How to exploit new hardware designs
  - Multicore, hardware performance monitors
- Resource-constrained devices (space, power …)
- Reducing complexity

Future Directions

- Better synergy with other levels of virtualization
  - App server, OS, low level virtualization
    - Eg. Hertz et al. '05
      - Extend garbage collector to be aware of paging
  - One level of indirection is clever, is > 1 redundancy?
- Better synergy with hardware
  - ISA is another level of virtualization!
    - Eg. Adl-Tabatabai et al. '04
      - Uses HW perf counter to drive prefetching optimization
- Additional focus on real-time performance, security, and reliability
  - Realtime eg: Bacon et al. [POPL'03, EMSOFT'05]
- Virtual machines for "static" languages, such as C, Fortran, etc. [Stoodley, CGO'06 Keynote]

Concluding Thoughts

- SE demands and processor frequency scaling issues require software optimization to deliver performance
- Virtual machines are here to stay
  - Independent of popular language of the day
- Dynamic languages require dynamic optimization
  - An opportunity for “dynamic” thinkers
- In many cases industrial practice is ahead of published research
- Still plenty of open problems to solve
- How can we encourage VM awareness in universities?

Additional Information – details on my web page

- 3-day Future of Virtual Execution Environments Workshop, Sept'04
  - 32 experts, hosted by IBM
  - Slides and video for most talk and discussion are available
- VEE Conference
  - VEE'07 will be co-located with FCRC/PLDI'07, June 13-15, San Diego
  - Submission Deadline: Feb 5, 2007
  - General Chair: Chandra Krintz (UCSB)
  - Co-program chairs: Steve Hand (Cambridge), Dave Tarditi (Microsoft)

Acknowledgements

- Toshio Suganuma for data and slides on IBM DK for Java
- AJ Shankar for data and slides
- Matthew Arnold, Steve Fink, and Dave Grove for feedback and significant material

General References

- “A Survey of Adaptive Optimization in Virtual Machines” by Arnold, Fink, Grove, Hind and Sweeney. Proceedings of IEEE, Feb 2005.
  - contains an extensive bibliography
- Advanced Compiler Design and Implementation by Muchnick. Published by Morgan Kaufmann, 1997.
- Engineering a Compiler by Cooper & Torczon. Published by Morgan Kaufmann 2004.

References: Case Study Virtual Machines

- Self
  - [Chambers&Ungar'91] "Making Pure Object-Oriented Languages Practical" by Chambers and Ungar. OOPSLA 1991.
  - [Hölzle&Ungar'94] "A Third Generation Self Implementation: Reconciling Responsiveness with Performance" by Hölzle and Ungar. OOPSLA 1994.
- Jikes RVM
  - [Arnold et al '00] "Adaptive Optimization in the Jalapeño JVM" by Arnold, Fink, Grove, Hind, and Sweeney, OOPSLA 2000.
  - [Fink et al. OOPSLA'02 tutorial] "The Design and Implementation of the Jikes RVM Optimizing Compiler" by Fink, Grove, and Hind.
  - [Alpern et al '05] IBM Systems Journal: historical overview
  - Lots of other pubs, see Jikes RVM web site
- IBM DK
  - [Suganuma et al '01] "Design and Evaluation of Dynamic Optimizations for a Java just-in-time" by Suganuma, Yasue, Kawahito, Komatsu, and Nakatani. TOPLAS'05.
  - Lots of other pubs, search for "Nakatani"

Referenced Papers

- A. Adl-Tabatabai, R.L. Hudson, M. J. Serrano, and S. Subramoney, ``Prefetch injection based on hardware monitoring and object metadata,'' PLDI'04.
- A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman, ``The StarJIT compiler: A dynamic compiler for managed runtime environments,'' Intel Technology Journal, 7(1) 19-31, Feb. 2003.
- B. Alpern, A. Cocchi, and D. Grove, "Dynamic type checking in Jalapeño," in *Proc. Usenix Java Virtual Machine Research and Technology Symp. (JVM'01)*, 41–52.
- G. Ammons, T. Ball, J. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling" PLDI'97, 85-96.
- M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the Jalapeño JVM," OOPSLA'00, *ACM SIGPLAN Notices*, 35(10), 47–65, Oct. 2000.
- M. Arnold and P.F. Sweeney, "Approximating the Calling Context Tree via Sampling",
IBM Technical Report: RC-21789, July, 2000.
- M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, *Architecture and Policy for Adaptive Optimization in Virtual Machines*, IBM Research Report #23429, Nov 12, 2004.

Referenced Papers

- M. Arnold, A. Welc, VT. Rajan, *Improving Virtual Machine Performance Using a Cross-Run Profile Repository*, OOPSLA'05.
- M. Arnold and D. Grove, Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines, CGO'05.
- M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," PLDI'01, *ACM SIGPLAN Notices*, 36(5), 168–179, May 2001.
- M. Arnold and B.G. Ryder, ``Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading,'' ECOOP'02.
- D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, ``Thin locks: Featherweight synchronization for Java, PLDI'98, ACM SIGPLAN Notices, 33(5), 258-268.
- D. F. Bacon, S. J. Fink, and D.Grove, ``Space- and time-efficient implementations of the Java object model,'' ECOOP'02.

Referenced Papers

- D. Bacon, P. Cheng, and V.T. Rajan, “A Real-time Garbage Collector with Low Overhead and Consistent Utilization”, POPL’03
- D. Bacon, P. Cheng, D. Grove, M. Hind, V.T. Rajan, E. Yahav, M. Hauswirth, C. Kirsch, D. Spoonhower, and M. Vechev, “High-level Real-time Programming in Java, EMSOFT’05.
- C. Click, “Global Code Motion/Global Value Numbering”, PLDI’95, 246-257.
- C. Chambers and D. Ungar, “Making pure object-oriented languages practical,” OOPSLA’91, 1–15.
- C. Chambers and D. Ungar, “Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs,” PLDI’00, *ACM SIGPLAN Notices*, 25(6), 150–164, Jun. 1990.
- C. Click and J. Rose, “Fast subtype checking in the HotSpot JVM,” in *Proc. Joint ACMJava Grande—ISCOPE 2001 Conf.*, 96–107.
- T. M. Chilimbi and J. R. Larus, ``Using generational garbage collection to implement cache-conscious data placement,'' PLDI’99, ACM SIGPLAN Notices, 34(3), 37--48, Mar. 1999.
- T. Chilimbi and R. Shaham, “Cache-conscious Coallocation of Hot Data Streams”, PLDI’06

Referenced Papers

- M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth, ``The open runtime platform: A flexible high-performance managed runtime environment,'' Intel Technology Journal, 7(1), 5--18, 2003.
- J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proc. 9th Eur. Conf. Object-Oriented Programming*, 1995, 77–101.
- J. Dean and C. Chambers, ``Towards better inlining decisions using inlining trials,'' LISP and Functional Programming, 1994, 273-282.
- D. Detlefs and O. Agesen, ``Inlining of virtual methods,'' ECOOP'99, 258-278.
- L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," POPL'84, 297–302.
- R. Dimpsey, R. Arora, and K.Kuiper, ``Java server performance: A case study of building efficient, scalable JVMs,'' IBM Systems Journal, 39(1), 151-174, Feb. 2000.
- P. C. Diniz and M. C. Rinard, "Dynamic feedback: An effective technique for adaptive computing." PLDI'97, ACM SIG}PLAN Notices, 32(5):71-84.

Referenced Papers

- S. J. Fink , D. Grove, and M. Hind, *The Design and Implementation of the Jikes RVM Optimizing Compiler,* OOPSLA'02 tutorial.
- S. J. Fink and F.Qian, ``Design, implementation and evaluation of adaptive recompilation with on-stack replacement,'' *CGO'03*, 241--252.
- G. Fursin, A. Cohen, M. O'Boyle, and O.Temam. "A practical method for quickly evaluating program optimizations." 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005), number 3793 in LNCS, pages 29-46. Springer Verlag, November 2005.
- Neal Glew, Spyros Triantafyllis, Michal Cierniak, Marsha Eng, Brian Lewis, and James Stichnoth. LIL: An Architecture-Neutral Lanugage for Virtual-Machine Stubs. In *3rd Virtual Machine Research and Technology Symposium*, San Jose, CA, USA, 111-125, May 2004.
- J. Gosling, B. Joy, and G. Steele, *The Java Language Specification.* Reading, MA: Addison-Wesley, 1996.

Referenced Papers

- G. J. Hansen, “Adaptive systems for the dynamic run-time optimization of programs,” Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1974.
- B. Hayes, ``Using key object opportunism to collect old objects,'' OOPSLA’91, 33-46 U. Hölzle, C. Chambers, and D. Ungar, “Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches, ECOOP’91
- M. Hertz, Y. Feng, E. Berger, “Garbage Collection Without Paging”, PLDI’05
- M. Hirzel, A. Diwan, and M. Hind, “Pointer Analysis in the Presence of Dynamic Class Loading”, ECOOP’04.
- U. Hölzle and D. Ungar, “A third generation SELF implementation: Reconciling responsiveness with performance,” OOPSLA’94, *ACMSIGPLAN Notices*, 29(10), 229–243, Oct. 1994.
- X. Huang, S. M. Blackburn, K. S. McKinley, J.E.B. Moss, Z.Wang, and P.Cheng, ``The garbage collection advantage: Improving program locality,'' OOPSLA’04.
- K. Ishizaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu, and T. Nakatani, “Effectiveness of cross-platform optimizations for a Java just-in-time compiler,” OOPSLA’03, *ACM SIGPLAN Notices*, 38(11), 187–204, Nov. 2003.

Referenced Papers

- R. Jones and R. Linz, Garbage collection: algorithms for automatic dynamic memory management, 1996.
- T. Kistler and M. Franz, ``Continuous program optimization: {A} case study,'' TOPLAS 25(4), 500-548. July 2003.
- K. Kawachiya, A. Koseki, and T. Onodera, “Lock reservation: Java locks can mostly do without atomic operations,'' *OOPSLA'02, ACM SIGPLAN* Notices, 37(11), 130-141, Nov. 2002,
- C. Krintz and B. Calder, *Using Annotation to Reduce Dynamic Optimization Time*, PLDI'01.
- C. Krintz, *Coupling On-Line and Off-Line Profile Information to Improve Program Performance*, CGO'03.
- J. Lau, M. Arnold, M. Hind, and B. Calder, “Online Performance Auditing: Using Hot Optimizations Without Getting Burned”, PLDI'06
- J. McCarthy, “History of LISP,” *ACM SIGPLAN Notices*, 13(8), 217–223, Aug. 1978.
- M. Paleczny, C. Vick, and C. Click, “The Java hotspot server compiler,” in *Proc. Usenix Java Virtual Machine Research and Technology Symp. (JVM'01)*, 2001, 1–12.

Referenced Papers

- K. Pettis and R.C. Hansen, ``Profile guided code positioning,'' PLDI'90 25(6), 16-27.
- T. Printezis, ``Hot-swapping between a Mark Sweep and a Mark & Compact garbage collector in a generational environment,'' JVM'01, Apr. 2001, 171--183.
- M. Serrano, R. Bordawekar, S.Midkiff, and M. Gupta, "Quicksilver: A quasistatic compiler for Java," OOPSLA'00, *ACM SIGPLAN Notices*, 35(10), 66–82, Oct. 2000.
- A. Shankar, S. Subramanya, R. Bodik, J. Smith, Runtime Specialization With Optimistic Heap Analysis, OOPSLA'05.
- D. Siegwart and M Hirzel, ISMM'06, "Improving Locality with Parallel Hierarchical Copying GC.
- M. Smith. "Overcoming the Challenges to Feedback-Directed Optimization," Dynamo'00, ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00), invited paper, Boston, MA, January 18, 2000. Also appears in ACM SIGPLAN Notices, 35(7):1–11, July 2000.
- S. Soman, C. Krintz, and D. Bacon, ``Dynamic selection of application-specific garbage collection,'' ISMM'04.
- S. Srinivas, Y. Wang, M. Chen, Q. Zhang, E. Lin, V. Ushakov, Y. Zach, S. Goldenberg, "*Java* JNI Bridge: An MRTE Framework for Mixed Native ISA Execution*", CGO'06.

Referenced Papers

- K. Stoodley, "*Productivity and Performance: Future Directions in Compilers*" CGO'06 Keynote
- M. Stoodley, *Challenges to Improving the Performance of Middleware Applications, MRE'05 talk.*
- L. Su and M. Lipasti, "Dynamic Class Hierarchy Mutation", CGO'06
- T. Suganuma, T. Yasue, T. Nakatani, "An Empirical Study of Method Inlining for a Java Just-in-Time Compiler", JVM'02.
- T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "Design and Evaluation of Dynamic Optimizations for a Java just-in-time", TOPLAS'05, 27(4), 732-785
- M. J. Voss and R.Eigemann. "High-level adaptive program optimization with ADAPT", PPOPP'01, ACM SIGPLAN Notices, 36(7):93—102.
- J. Whaley, "A portable sampling-based profiler for Java virtual machines", Java Grande 2000, 78-87.
- J. Whaley, ``Partial method compilation using dynamic profile information,'' OOPSLA'01, SIGPLAN Notices, 36(11), 166--179, Nov. 2001.
- X. Zhuang, M. Serrano, H. Cain, J-D. Choi, Accurate, Efficient, and Adaptive Calling Context Profiling, PLDI'06.