Dynamic Compilation and Adaptive Optimization in Virtual Machines

Instructor: Michael Hind

Material contributed by: Matthew Arnold, Stephen Fink, David Grove, and Michael Hind
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 \(\rightarrow\) 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 ($x \ll X$)

- 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 ($x \ll X$)

- Computer architects providing multicore machines
  - Will require software to utilize these resources
  - Not clear if it will contribute more than in the past ($y \approx Y$)

- 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
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

IA32 binary

Machine Dependent

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 $\rightarrow$ 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 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

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.)

Steady State

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

```java
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

```
A   A   A   B   B   C   ...   A   B   C   ...   A   A
B   B   C   ...   B   C   ...   B   B
```
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

Sample
How to Find Candidates for Optimization

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

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

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 interpretation
  - 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

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>D</td>
</tr>
</tbody>
</table>

- 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

```
If (dispatch target is foo')

invokevirtual foo

inlined foo

invokevirtual foo
```
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 \(\rightarrow\) better forward dataflow

```
invokevirtual foo
```

```
inlined foo
```

```
Trigger OSR
```

```
No-op
```

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

```java
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 $m_1$ to compiled code $m_2$ even while $m_1$ runs on some thread’s stack

Extremely general mechanism ➔ minimal restrictions on speculation
OSR Mechanisms

- Extract *compiler-independent state* from a suspended activation for $m_1$
- Generate new code $m_2$ for the suspended activation
- Transfer execution to the new code $m_2$
OSR and Inlining

Suppose optimizer inlines $A \rightarrow B \rightarrow C$: 

1. JVM Scope Descriptor $A$
2. JVM Scope Descriptor $B$
3. JVM Scope Descriptor $C$

stack

A

frame

PC

A'

B'

C'

frame

frame

frame

PC
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
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 ...
```

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

- Rectangle code
- Circle code

**PIC stub**

```
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]
  
  ![Diagram of tri-state hash code encoding]

- Can also elide lockword [Bacon et al.'02]
  
  ![Diagram showing hashcode and lockword statuses]
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
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 4\textsuperscript{th} Wave of Adaptive Optimization?

- Try multiple optimization strategies for a code region, \textit{online}
- Run and time all versions online
- Determine which performs the best
- Use it in the future

\textbf{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]
  - \textit{More details to follow}
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?