Dynamic Compilation and Adaptive Optimization in Virtual Machines

Stephen Fink, David Grove, and Michael Hind

Who are we?

- Work for IBM, home of 2 other Java VMs
- In previous lives, worked on static whole program analysis and optimization for C, C++, Cecil, Fortran, Java, Modula-3, Smalltalk
- Excited to share what we know
Tutorial Goals

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

True or False?

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. A static compiler will always produce better code than a dynamic compiler.
4. Sophisticated profiling is too expensive to perform online.
5. Program optimization is a dead field.
6. Small academic research groups cannot afford the infrastructure investment to innovate in this field.

All of these myths are false!!!
Tutorial Outline

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

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

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Compiler(s)</th>
<th>Adaptive Optimization System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Management</td>
<td>Libraries</td>
<td>Security Services</td>
</tr>
<tr>
<td>Tracing, Profiling, etc. (ex. JVMPI)</td>
<td>Runtime Support Mechanisms</td>
<td>Dynamic type checking Introspection, etc.</td>
</tr>
</tbody>
</table>

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

- **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
Quick History of VMs

- Java/JVM [Gosling et al '95]
  - 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
  - Much of the technology was transferred to Sun’s HotSpot JVM

- IBM DK for Java ['95-'04]
  - Port of Sun Classic JVM + JIT + GC and synch enhancements
  - Compliant JVM
  - World class performance

- Jikes RVM (Jalapeno) ['97-'04]
  - VM for Java, written in (mostly) Java
  - Independently developed VM + GNU Classpath libs
  - Open source, popular with researchers, not a full JVM
Tutorial Outline

1. Background

2. Adaptive Optimization
   • Selective Optimization
     - Designing an adaptive system
     - Profiling mechanisms and recompilation policies
     - Case studies: Jikes RVM and IBM DK for Java

3. Engineering a JIT Compiler

4. Feedback-Directed and Speculative Optimizations

5. Wrapping Up

How are Programs Executed?

1. Interpretation
   - Popular approach for high-level languages
     - Ex, APL, SNOBOL, BCPL, Perl, Python, MATLAB
   - Useful for memory-challenged environments
   - Low startup overhead, but much slower than native code execution

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 myths
Interpretation vs. (Dynamic) Compilation

Example: 500 methods
Overhead: 20x
- Interpreter: 0.01 time units/method
- Compilation: 0.20 time units/method

Execution: Compiler gives 4x speedup

Selective Optimization

- Hypothesis: most execution is spent in a small pct. of methods

- Idea: use two execution strategies
  1. Interpreter or non-optimizing compiler
  2. Full-fledged optimizing compiler

- Strategy:
  - Use option 1 for initial execution of all methods
  - Profile application to find "hot" subset of methods
  - Use option 2 on this subset
Selective Optimization

Selective opt: compiles 20% of methods, representing 99% of execution time

Designing an Adaptive Optimization System

- What is the system architecture?
- What is the mechanism (profiling) and policy for driving recompilation?
  - How effective are these systems?
General Architecture

- Executing Program
  - Optimized/Instrumented Code
- Runtime Measurements
  - Processed Profile Data
- Controller
  - Compilation Decisions
- Recompilation Subsystem
  - Raw Profile Data
- History
  - Measurements

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 JRocket
- CLR
  - Multiple compilers
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 and 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

Sample
How to Find Candidates for Optimization

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

```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
Case Studies

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

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
  - \( \text{cur} \), current opt level for method \( m \)
  - \( \text{Exe}(j) \), expected future execution time at level \( j \)
  - \( \text{Comp}(j) \), compilation cost at opt level \( j \)
- Choose \( j > \text{cur} \) that minimizes \( \text{Exe}(j) + \text{Comp}(j) \)

- If \( \text{Exe}(j) + \text{Comp}(j) < \text{Exe}(\text{cur}) \) 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

Startup Programs: Jikes RVM

![Speedup over Baseline](chart)

- \( \text{No FDO, Mar'04, AIX/PPC} \)
Startup Programs: Jikes RVM

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

IBM DK for Java [Suganuma et al. ’01]

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]
Understanding System Behavior

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

Other issues

- Synchronous vs. asynchronous recompilation
  - Is optimization performed in the background?
- Static or dynamic view of profile data
  - Is profile data packaged or used in flight?
- Skipping optimization levels
  - How to decide when to do it?
- Collecting dead compiled code
  - When is it safe?
- Installing new compiled code
  - Stack rewriting, code patching, etc.
- RAS issues
  - How repeatable?
Tutorial Outline

1. Background
2. Adaptive Optimization
3. Engineering a JIT compiler
   • What is a JIT compiler?
   • Case studies of 3 Java JITs
   • High level language-specific optimizations
   • VM/JIT interactions
4. Feedback-Directed and Speculative Optimizations
5. Wrapping Up

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 true in the presence of (sampling-based) feedback-directed optimization
  - So far, a bias to performance at the expense of RAS, but that is changing as VM technology matures

Structure of a JIT compiler

bytecode

Front-end

Common Optimizer

Machine Dependent

Machine Dependent

IA32 binary

PPC/32 binary
Are JITs really that different from traditional compilers?

- Look at three example Java JITs
  - Jikes RVM
  - IBM DK
  - HotSpot

- High level language-specific optimizations

- Rich bi-directional VM/JIT interface

---

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 O0 optimizations
- More sophisticated code reordering algorithm [Pettis&Hansen]

- Over time many optimizations shifted from O1 to O0
- Aggressive inlining is currently the primary difference between O0 and O1
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)

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

Cost Effectiveness of Optimizations in IBM DK

- 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
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 (thus 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.
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

```java
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);
}
```

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

JIT/VM Interactions

- Runtime services often require support from JIT
  - Memory management
  - Exception delivery and symbolic debugging

- JIT generated code assumes 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 JVM’s
  - 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

Tutorial Outline

1. Background
2. Adaptive Optimization
3. Engineering a JIT Compiler
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 the VM
5. Wrapping Up
Feedback-Directed Optimization (FDO)

- Exploit information gathered at run-time to optimize execution
  - "selective optimization": what to optimize
  - "FDO": how to optimize

- Advantages of FDO [Smith 2000]
  - Can exploit dynamic information that cannot be inferred statically
  - System can change and revert decisions when conditions change
  - Runtime binding allows more flexible systems

- Challenges for fully automatic online FDO
  - Compensate for profiling overhead
  - Compensate for runtime transformation overhead
  - Account for partial profile available and changing conditions

FDO Can Improve Performance in Long Running Applications: Jikes RVM (Mar ’04)
Challenge: Don’t Degrade Performance Too Much When Collecting Profile Data [Arnold et al. ’02]

Profiling

- 4 Categories
  - Runtime service monitors
    - e.g. dispatch tables, synchronization services, GC
  - Hardware performance monitors
    - see Adl-Tabatabai et al., Friday 9AM
  - Sampling
    - e.g. sample method running, call stack at context switch
  - Program instrumentation
    - e.g. basic block counters, value profiling
- Myth: Sophisticated profiling is too expensive to perform online
- Reality: Well-known technology can collect sophisticated profiles with sampling and minimal overhead
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

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
- Adopted by IBM J9 VM
Common FDO Techniques

- Compiler optimizations
  - Inlining
  - Code Layout
  - Multiversioning
  - Potpourri
- Run-time 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>A</td>
<td>300</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></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]

![Code positioning diagram](image)

### 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 run-time 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 run-time (more later)
- No downstream merge -> better forward dataflow

```
No-op
```

```
invokevirtual foo
```

inlined foo'

```
Trigger OSR
```

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
  - Naive 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.
- For desirable specializations, compiler dynamically installs instrumentation to do 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
    - CFA Precision Improvement \( \rightarrow \) 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]

<table>
<thead>
<tr>
<th>Relative Performance to No Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMI-full</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mtrt</th>
<th>jess</th>
<th>compress</th>
<th>db</th>
<th>mpegaudio</th>
<th>jack</th>
<th>javac</th>
<th>SPECjbb</th>
<th>Geo. Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>3.8</td>
<td>3.4</td>
<td>3.1</td>
<td>3.0</td>
<td>2.5</td>
<td>2.4</td>
<td>2.3</td>
<td>2.1</td>
</tr>
</tbody>
</table>

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 only escape on cold paths
- Software prefetching
  - Profile data guides placement of prefetch instructions
Aggressive Speculation and Invalidation Techniques

- Speculative code generation
  - Generate code that would be incorrect if some condition changes
  - Invalidate generated code to recover if needed

- Why speculate?
  - Hard to analyze features (reflection, native code, classloading)
  - Heavier usage of OO language features, generic frameworks
  - Constraints on compilation resources

- How to invalidate speculative code?
  - On-Stack Replacement (OSR)
  - Pre-existence

Invalidation via On-Stack Replacement (OSR)

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
Applications of OSR

- Deferred compilation [SELF-91, HotSpot, Whaley 2001]
  - Don’t compile uncommon cases
  - Improve dataflow optimization and reduce compile-time
- Runtime optimization of long-running activations [SELF-93]
  - Promote long-running loops to higher optimization level
- Debug optimized code via dynamic deoptimization [HCU92]
  - At breakpoint, deoptimize activation to recover program state
- Safe invalidation for speculative optimization
  - Class-hierarchy-based inlining [HotSpot]
  - Type prediction [SELF-91]
  - Escape Analysis [Whaley 2001]

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 $A \to B \to C$:

![Diagram showing stack and frames]

**OSR Challenges**

- **Engineering Complexity**
  - Need to capture/recover program state without constraining optimizations or introducing runtime overhead to the point where the optimization benefits are lost
  - Retro-fitting to existing systems (especially JITs) can be quite complex
  - Support for multi-threading, weak memory model SMPs, etc.
- **RAS implications**
  - Code that is both complex and infrequently executed is a prime location for bugs
Invalidation via pre-existence [Detlefs & Agesen'99]

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

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

- **Key insight:** if inlining m1 without a run-time 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 only required for all future invocations
  - No interrupted activations a la OSR

---

Tutorial Outline

1. Background
2. Adaptive Optimization
3. Engineering a JIT Compiler
4. Feedback-Directed and Speculative Optimizations
   - Gathering profile information
   - Exploiting profile information in a JIT
   - Exploiting profile information in the VM
      - Dispatch optimizations
      - Speculative object models
      - GC and locality optimizations
5. Wrapping Up
Virtual/Interface Dispatch

- Polymorphic inline cache [Holzle et. al 91]

```plaintext
... receiver = ...
call PIC stub ...
```

**Calling code**

**PIC stub**

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

**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 98, Agesen Sun EVM]

  ```plaintext
  Unhashed
  Hashed
  (hashcode == address)
  Hashed and Moved
  ```

- Can also elide lockword [Bacon et. el 2002]

  ```plaintext
  Has synchronized method
  No synchronized method
  No synchronized method, but locked
  ```
Adaptive GC techniques

- Dynamically adjust heap size
  - IBM DK [Dimpsey et al. 2000] – policy depends on heap utilization and fraction of time spent in GC
- Switch GC algorithms to adjust to application behavior
  - [Printezis 2001] – switch between Mark&Sweep and Mark&Compact for mature space in generational collector
  - [Soman et al. 2003] – more radical approach prototyped in Jikes RVM
    - Not yet exploited in production VMs
- Opportunistic GC
  - [Hayes 1991] – key objects keep large data structures live
    - Not yet exploited in production VMs

Spatial Locality Optimizations [e.g. Kistler, Chilimbi]

- Move objects or change objects to increase locality
  - Field reordering
  - Object splitting
  - Object co-location

- Encouraging academic results, mostly with offline profiling
- Not yet (to our knowledge) fruitful in production VMs
Tutorial Outline

1. Background
2. Adaptive Optimization
3. Engineering a JIT Compiler
4. Feedback-Directed and Speculative Optimizations
5. Wrapping Up
   - Comparison of HLL VMs and dynamic binary optimizers
   - Myths revisited and areas for future research

Comparison between HLL VMs and Dynamic Binary Optimizers

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

Trends suggest that more programs will be written to managed HLLs
   - Open question: for managed HLL, does binary optimizer add value?
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: 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 IV

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 V

Myth: Program optimization is a dead field, law of diminishing returns

Reality:
  • No shortage of research topics
    - FDO
      - What optimizations should be performed?
      - How should a VM use FDO?
      - When should a VM switch from one form of FDO to another because of a program phase change?
    - Theoretically-grounded selective optimization
    - Empirical optimization policies
    - Deep online/staged analysis
    - Higher-level programming models (e.g. J2EE, ASP.NET, Web Services, BPEL)
    - Resource-constrained devices (space, power ...)

Myths Revisited VI

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
      http://orp.sourceforge.net
    - Mono
      http://go-mono.com
    - Insert your favorite infrastructure here.
Acknowledgements

- Toshio Suganuma for data and slides on IBM DK for Java
- Matthew Arnold for slides and general feedback
- Tamiya Onodera for feedback

General References

- 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**

- **Jikes RVM**

- **IBM DK**
Code Size Comparison, startup: IBM DK for Java

![Bar chart showing code size comparison]

Code Size Comparison, steady state: IBM DK for Java

![Bar chart showing code size comparison]
Execution Profile: Jikes RVM (Jul '02)

Size 100, SPECjvm98, 1 run each

- Application Threads: 85.6%
- Garbage Collection: 6.5%
- Controller: 0.1%
- Method organizer: 0.0%
- Decay organizer: 0.0%
- Inlining organizer: 0.0%
- Opt. Recompilation: 0.6%

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)

![Graph showing recompilation activity](image)

Static Multiversioning: Splitting

- Often used in conjunction with guarded virtual inlining

```
If (c)
  A  B
  If (c)
    D  E
```

```
If (c)
  A  B
  If (c)
    D  E
```
Static multiversioning: Loop Versioning

```java
for (int i = 0; i < j; i++) {
    try {
        sum += A[i];
    } catch (ArrayOutOfBoundsException e) {
        die();
    }
}
```

If \( j < A.length \)

```java
for (int i = 0; i < j; i++) {
    sum += A[i];
}
```