Just-In-Time and Dynamic Compilation Techniques

Kostis Sagonas
kostis@it.uu.se

Lecture's Outline

1. Background
2. Selective and Adaptive Compilation
3. JIT Compiler Engineering
4. Feedback-directed and Speculative Optimizations

Terminology: Virtual Machine

Virtual machine (VM) is a software execution engine for a program written in a machine-independent language

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

How are Programs Executed?

1. Interpretation
   - Popular approach for high-level languages
     - Ex. APL, 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 function to native code on first invocation
     - Ex. ParcPlace Smalltalk-80, Self-91
     - Initial high (time & space) overhead for each compilation
     - Sophisticated optimizations (e.g., SSA, etc.) typically not performed due to their (perceived) high cost
   - Responsible for many myths

Lecture's Outline

1. Background
2. Selective and Adaptive Compilation
   - What is selective and adaptive compilation?
   - How to find candidates?
   - How to decide what to recompile?
   - Case studies
3. JIT Compiler Engineering
4. Feedback-directed and Speculative Optimizations

Selective and Adaptive Optimization

Hypothesis: Most execution is spent in a small percentage of functions/code

Idea: use two execution strategies:

1. Interpreter or non-optimizing compiler
2. Full-fledged optimizing compiler

Approach:

- Use strategy 1 for initial execution of all functions
- Profile application to find "hot" subset of functions
- Use strategy 2 for this subset

Selective Optimization Examples

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

Profiling: Finding Candidates for Optimization

- Counters
- Call stack sampling
- Combinations
  - E.g., use counters initially and sampling later on
  - Ex. IBM DK for Java

Profiling via Counters

- Insert function-specific counters on function entry and loop back edges
- Count how often a function is called and approximate how much time is spent in the function
- Very popular approach: Self, Hotspot Java, …
- Issue: Overhead for incrementing counters might be significant
  - Not present in original code

Profiling via Call Stack Sampling

- Periodically record which function(s) are on the call stack
- Approximates amount of time spent in each function
- Does not necessarily need to be compiled into the code
  - Ex. Jikes RVM:
    - samples occur at taken yield points (approx 100/sec)
    - organizer thread communicates sampled methods to controller
- Issue: timer-based profiling is not deterministic

Recompilation Policies

Problem: Given recompilation candidates, which ones should be optimized?

Counters:

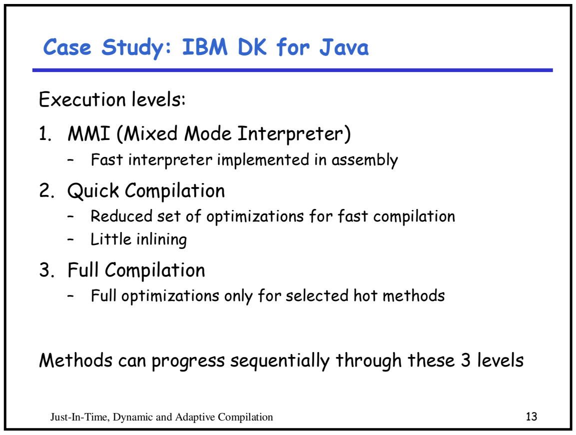
1. Optimize function that surpasses threshold
   - Simple but hard to tune; doesn't consider context
2. Optimize function on the call stack based on inlining policies
   - Addresses context issue

Call Stack Sampling:

1. Optimize all functions that are sampled
   - Simple but doesn't consider frequency of sampled functions
2. Use a cost/benefit model (Jikes RVM)
   - Seemingly complicated but easy to engineer
   - Maintenance free
   - Naturally supports multiple optimization levels

The Cost/Benefit Model of Jikes RVM

- Define
  - cur: current optimization level of method m
  - Exe(j): expected future execution time if compiled at level j
  - Comp(j): expected compilation cost at optimization level j
- Choose $j > cur$ that minimizes Exe(j) + Comp(j)
- If Exe(j) + Comp(j) < Exe(cur) then recompile at level j
- Assumptions:
  - Sample data determines how long a method has executed
  - Method will continue to execute as much in the future as it has in the past
  - Compilation speed and speedup are offline averages

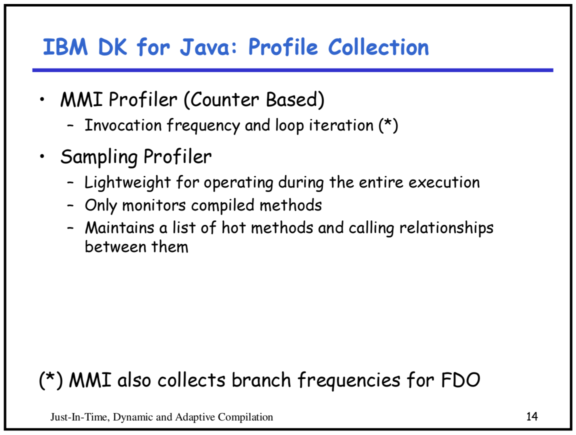


Case Study: IBM DK for Java

Execution levels:

1. MMI (Mixed Mode Interpreter)
   - Fast interpreter implemented in assembly
2. Quick Compilation
   - Reduced set of optimizations for fast compilation
   - Little inlining
3. Full Compilation
   - Full optimizations only for selected hot methods

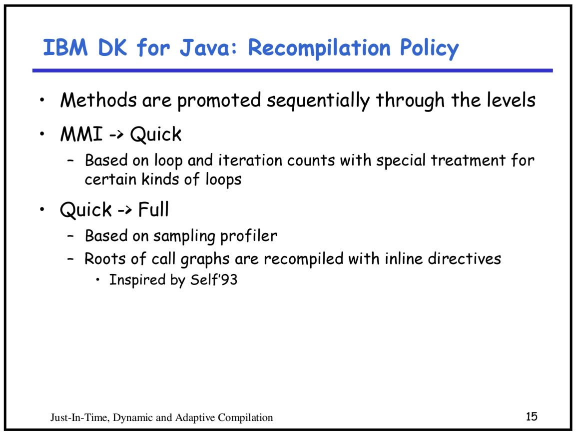
Methods can progress sequentially through these 3 levels



IBM DK for Java: Profile Collection

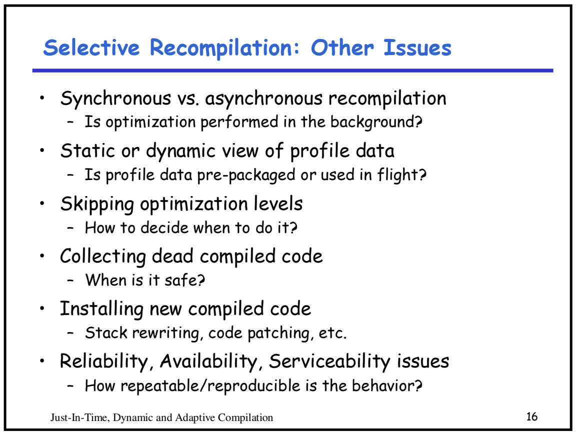
- MMI Profiler (Counter Based)
  - Invocation frequency and loop iteration (*)
- Sampling Profiler
  - Lightweight for operating during the entire execution
  - Only monitors compiled methods
  - Maintains a list of hot methods and calling relationships between them

(*) MMI also collects branch frequencies for FDO



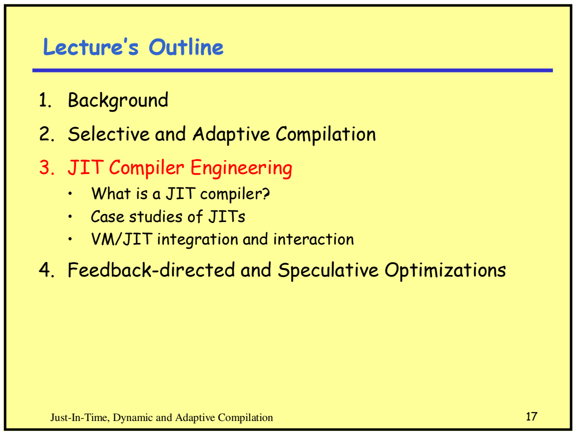
IBM DK for Java: Recompilation Policy

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



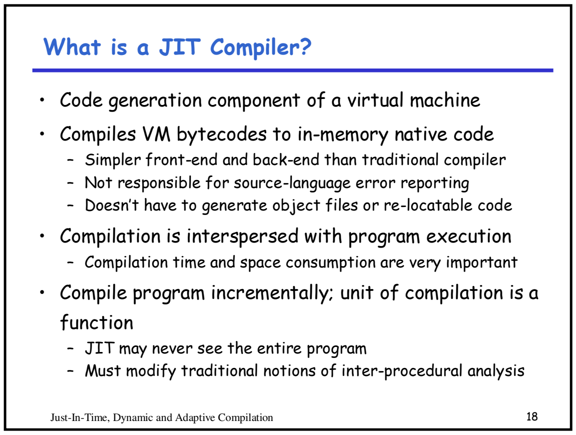
Selective Recompilation: Other Issues

- Synchronous vs. asynchronous recompilation
  - Is optimization performed in the background?
- Static or dynamic view of profile data
  - Is profile data pre-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.
- Reliability, Availability, Serviceability issues
  - How repeatable/reproducible is the behavior?



Lecture's Outline

1. Background
2. Selective and Adaptive Compilation
3. JIT Compiler Engineering
   - What is a JIT compiler?
   - Case studies of JITs
   - VM/JIT integration and interaction
4. Feedback-directed and Speculative Optimizations



What is a JIT Compiler?

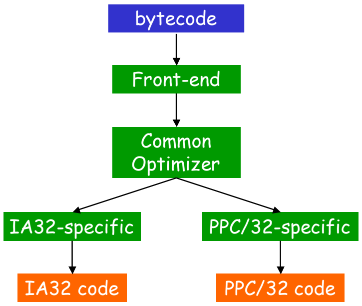
- Code generation component of a virtual machine
- Compiles VM bytecodes to in-memory native code
  - Simpler front-end and back-end than traditional compiler
  - Not responsible for source-language error reporting
  - Doesn't have to generate object files or re-locatable code
- Compilation is interspersed with program execution
  - Compilation time and space consumption are very important
- Compile program incrementally; unit of compilation is a function
  - JIT may never see the entire program
  - Must modify traditional notions of inter-procedural analysis

JIT Compiler: Design Requirements

- High performance (of executing application)
  - Generate "reasonable" code at "reasonable" compilation times
  - Selective optimization enables multiple design points
- Deployed on production servers
  - Reliability, Availability, and Serviceability (RAS) requirements
  - Facilities for logging and "replaying" compilation activity
- Tension between high-performance and RAS
  - Especially true in the presence of (sampling-based) feedback-directed optimizations
  - So far, a bias to performance at the expense of RAS, but that is changing as VM technology matures

---

Structure of a JIT Compiler (Example)



Case Study 1: Jikes RVM

- Java bytecodes ⇒ IA32, PPC/32
- 3 Intermediate Representations (IR)
  - All 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 compiler + 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 the 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 optimizations
- 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
- Runs multiple passes of many -O0 optimizations
- More sophisticated code reordering algorithm

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

Jikes RVM: Opt Level 2

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

Case Study 2: IBM DK

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

IBM DK: Optimizations on Extended Bytecodes

Java bytecodes + type information:
  - Compact representation
  - Can't express some transformations

- Flow-sensitive type inference (de-virtualization)
- Method inlining, includes guarded inlining
- Null-check and array bounds check elimination
- Flow-sensitive type inference (checkcast/instanceof)

IBM DK: Optimizations on Quadruples

Quadruples:
  - Register-based; CFG of extended basic blocks
  - Close to native instruction set; some pseudo-operations

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

IBM DK: Optimizations on DAG of QUADs

DAG: augment quadruples with explicit dependence edges

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

IBM DK: Cost Effectiveness of Optimizations

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

Case Study 3: HotSpot Server JIT

- HotSpot Server Compiler
  - Client compiler is simpler; small set of optimizations but faster compile time
- Java bytecodes ⇒ SPARC, IA32
- Extensive use of On Stack Replacement (OSR)
  - Supports a variety of speculative optimizations
  - Integral part of JIT's design
- Of the 3 JITs, it has the most advanced static optimizer
  - SSA form and heavy optimization
  - Design assumes selective optimization (thus HotSpot)

HotSpot Server JIT

- Virtually all optimizations done on SSA-based CFG
  - Global value numbering
  - Sparse conditional constant propagation
  - Fast/Slow path separation
  - Instruction selection
  - Global code motion
- Graph coloring register allocation with live-range splitting
  - Approx 50% of compile time
  - However, much more than just allocation
    - Out-of-SSA transformation, GC maps, OSR support, etc.

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 tests, 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
  - Can constrain some optimizations
- 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 insert 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 a Java VM 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 of 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
- Null pointer checks & array bounds checks
  - 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

JIT/VM Integration

- Integrating a JIT system where native code can coexist with interpreted code in the VM is not trivial
- Context switches between native and interpreted code have to be fast
  - They can occur at function calls, returns, and when exceptions are thrown
- Ensuring proper tail-calls with a mixed mode of execution is tricky

Lecture's Outline

1. Background
2. Selective and Adaptive Compilation
3. JIT Compiler Engineering
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
     - Dispatch optimizations
     - Adaptive GC techniques and locality optimizations

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
  - 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

Profiling Methods

Categories:

1. Runtime service monitors
   - E.g. dispatch tables, synchronization services, GC
2. Hardware performance monitors
3. Sampling
   - E.g. sample function running, call stack at context switch
4. 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

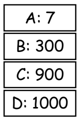
Common FDO Techniques

- Compiler optimizations
  - Inlining
  - Code layout (Code positioning)
  - Multiversioning
  - FDO 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 by sampling-based recomputation
- When sampling counter triggers, crawl up the stack to find "root" method of inline sequence



- D exceeds counter threshold
- Crawl up the stack to examine counters
- Recompile B and inline C and D

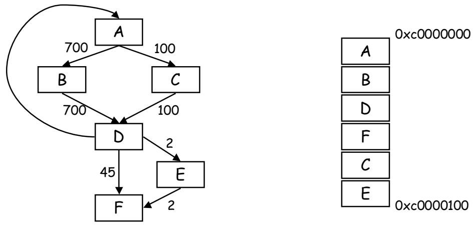
Fully Automatic Profile-Directed Inlining

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 call sites in "hot" methods
- Constructs partial dynamic call graph of "hot" call edges
- Inlining database to avoid performance perturbation
- Experimental conclusion
  - Use static heuristics for small size methods
  - Inline medium and bigger methods based on profile data

Code Positioning

- Easy and profitable: employed on most (all?) production VMs
- Synergy with trace scheduling



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

FDO Potpourri

- Many opportunities to use profile info during various compiler phases
- Almost any heuristic-based decision can be improved 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 in "cold" paths
  - Software prefetching

Aggressive Speculation

- 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, dynamic class loading)
  - Heavier use of OO language features, generic frameworks
  - Constraints on compilation resources
- How to invalidate speculative code?