Why Software Optimization Matters
and Some Thoughts on How to Improve it

David Grove

With material from Arnold, Fink, Grove, and Hind
CGO’04, PLDI’04, ETAPS’05 tutorials, CGO’05 keynote, and university seminars

May 16, 2005 UT Austin Seminar
Agenda

- Why Software Optimization Matters
- Debunking Dynamic Optimization Myths
- Conclusions
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 can pose performance challenges
  - Dynamic memory allocation
    - Need pointer knowledge to avoid conservative dependences
  - Object-oriented programming
    - Need efficient virtual dispatch, overcome small methods, extra indirection
  - Automatic memory management
    - Need efficient allocation and garbage collection algorithms
  - Runtime bindings
    - Need to deal with unknown information
  - ...

- Features require a rich runtime environment ➔ virtual machine
Type Safe, OO, VM-implemented Languages Are Mainstream

- **Java is ubiquitous**
  - eg. Hundreds of IBM products are written in Java

- **“Very dynamic” languages are widespread and run on a VM**
  - eg. Perl, Python, PHP, etc.

- **These languages are not just for traditional applications**
  - Virtual Machine implementation, eg. Jikes RVM
  - Operating Systems, eg. Singularity
  - Real-time and embedded systems, eg. Metronome-enabled systems
  - Massively parallel systems, eg. DARPA-supported efforts at IBM and Sun

- **Virtualization is everywhere**
  - browsers, databases, O/S, binary translators, hypervisors, 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 towards 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
    - Sevitzky, Mitchell, Srinivasan report
      - J2EE benchmark creates 10 business objects (w/ 6 fields) from a SOAP message of bytes
        - 10,953 calls, 1,492 objects created
  - Traditional compiler optimization wouldn’t help much
    - Optimization at a higher semantic level could yield 10x
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 ($z > Z$)
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 \((z > Z)\)
Agenda

- Why Software Optimization Matters
- Debunking Dynamic Optimization Myths
- Conclusions
Fact or Fiction?

1. Production VM’s avoid complex optimizations, favoring stability over performance
2. Because they execute at runtime, dynamic compilers must be blazingly fast
3. Dynamic class loading is a fundamental roadblock to cross-method optimization
4. Sophisticated profiling is too expensive to perform online
5. A static compiler will always produce better code than a dynamic compiler
6. Infrastructure requirements stifle innovation in this field
Myth 1 - Production VM’s 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 JVM’s perform many complex optimizations, including
  - Optimizations that require sophisticated coding
  - Difficult to debug dynamic behavior
    - Eg, 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 VM’s 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
    - Examples given throughout rest of talk
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
Myth 2 - *Dynamic compilers must be blazingly fast*

**Perception:** Dynamic compilers cannot spend time performing sophisticated optimizations

**Reality:** Production JITs look very similar to traditional compilers
- Perform all the classical optimizations
  - SSA, aggressive method inlining, graph-coloring register allocation (Hotspot), etc.
  - Multiple optimization levels, etc
- Selective optimization strategies successfully focus compilation effort where needed
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
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
Primary Tasks for Selective Optimization

1. Identify “hot” methods
   - Counters, e.g., Self, HotSpot, ORP, etc.
     - Count each method’s entry and (optionally) loop iterations
   - Call Stack Sampling, e.g., Jikes RVM, JRockit, etc.
     - Periodically record method(s) on top of call stack

2. Deciding when (and how) to optimize the hot methods
   - Optimize when count surpasses threshold
     - Most common approach
       - Simple to implement, works reasonably well
       - Tuning threshold values is difficult
       - Often leads to surprises for new benchmarks
   - Use cost/benefit model (Jikes RVM)
     - Seemingly complicated, but easy to engineer
     - System requires no “tuning”
Jikes RVM Architecture [Arnold et al. '00, '04]
Selective Optimization Effectiveness:
Jikes RVM, [Arnold et al., TR Nov’04]

- **Startup**
  - Speedup: Geometric mean of 12 benchmarks run with 2 different size inputs (SPECjvm98, SPECjbb2000, etc.)

- **Steady State**
  - Speedup: 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]

**Startup**

- 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)
But the world is not always simple

- Modern programs execute a large number of methods
- SPECjappserver, Mark Stoodley (IBM) MRE’05
  - executes > 10,000 methods
  - No single “hot spot”
    - Hottest method may be <1% of total execution time
  - 90/10 rule may still apply
    - But 10% of 10,000 is 1,000 (luke warm) methods
But the world is not always simple

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

- Eclipse startup, IBM J9 VM

<table>
<thead>
<tr>
<th>Workspace</th>
<th>Running Time</th>
<th>Number of Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>5.8 secs</td>
<td>10,499 740 (7.1%) 4 (0.04%)</td>
</tr>
<tr>
<td>Eclipse source</td>
<td>18.2 secs</td>
<td>18,960 2,169 (11.4%) 21 (0.11%)</td>
</tr>
</tbody>
</table>
**Example:** Jikes RVM Compilers on AIX/PPC

- Both efficiency and code quality of optimization are relevant
  - Improving the efficiency of optimization has value
  - Improving code quality has value
    - Even if expensive, can likely be incorporated via selective optimization
Research issues for selective optimization

- Tuning thresholds is a problem
  - Threshold values often turn out to be bad later on
  - Dealing with combined counter and sample data

- Pause times
  - Model optimizes throughput, ignores pauses
    - After running for an hour, may suggest massive compilations

- Synchronous vs. asynchronous recompiilation
  - Is optimization performed in the background?
  - Exploit idle CPU's
    - Dozens of compilations in parallel (Azul Systems)

- Handling programs with “flat” profiles
  - Use partial method compilation?

- Skipping optimization levels
  - When to do it?
  - Better ways to predict how long method will run?

- Reliability
  - Repeatability
  - Counters have advantages, and disadvantages
Myth III - Dynamic class loading is a fundamental roadblock to cross-method optimization

Perception: Because you never have the whole program, interprocedural optimizations are not possible ex) virtual method resolution, virtual inlining, escape analysis, etc.

Reality:

- Can perform speculative optimization
  - Generate code that is correct while certain conditions hold
  - Invalidate generated code to recover if needed
- Sophisticated invalidation technology well-understood; mitigates need for overly conservative assumptions
- Speculative optimization can be more aggressive than conservative, static compilation
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 $\Rightarrow$ minimal restrictions on speculation
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 [HCU92]
   - 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
Invalidation Discussion

- **OSR challenges**
  - Nontrivial to engineer
  - Keeping around extra state may introduce overhead

- **Other existing invalidation techniques**
  - Pre-existence inlining [Detlefs & Agesen'99]
  - Code patching [Suganama’02]
  - Thin Guards [Arnold’02]

- **Once invalidation mechanism exists**
  - Relatively easy to perform speculative optimizations
  - Many researchers avoid interprocedural analysis of Java for the wrong reasons
    - Invalidation is “easy”. The fun parts are
      - Must be able to detect when assumptions change
      - Must be low overhead, incremental
    - Area mostly unexplored (Hirzel et al, ECOOP’04)
Myth IV

Myth: A static compiler will give better performance than a dynamic compiler
  • Static compiler can use an unlimited amount of analysis time

Reality:
  • Many production JITs implement all classical static optimization
  • Two main advantages for a dynamic compiler
    1. Enables *Feedback-directed optimization*, which can be more effective than unlimited static analysis
    2. Dynamic languages (Java) are difficult to compile statically
       - Too much information is known only at runtime
Feedback-Directed Optimization (FDO)

- Exploit information gathered at runtime 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 has advantages

- 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
I Don't Buy the Hype. I want Static Compilation

- Despite the efforts of selective optimization
  - Static (ahead-of-time) compilation has advantages
    - Fast startup of large apps
  - CLR is taking this approach
    - Philosophy: quick startup and deterministic behavior is good
  - IBM J9 VM supports AOT compilation for Java
    - Mainly targeted at embedded and real-time applications

- There is a deeper question: do you want a dynamic language with open-world semantics?
  - Dynamic languages (Java) are difficult to compile statically
    - Too much information is known only at runtime
    - Quality of code generated by online JIT is vastly superior

- Dynamic features have a cost; are they worth it?
  - For many “real” Java applications (Eclipse, J2EE)
    - Class loading and reflection are relied upon heavily
Myth V

**Myth:** Sophisticated profiling is too expensive to perform online

**Reality:**
- If obtaining profile X would help improve performance, it can be done
- Sampling-based profiling is cheap and can collect detailed profile information
  - e.g. IBM DK dynamic instrumentation
  - e.g. Arnold-Ryder full-duplication framework
IBM DK Profiler [Suganuma et al '01,'02]

- Start with timer sampling
  - Used to identify methods for re-optimization
- Then use **Dynamic instrumentation**: finer granularity profile data
  1. *Patch* method entry with jump to instrumented version
  2. Run until threshold
     - Time bound
     - Desired quantity of data collected
  3. Undo patch

```
sub esp, 60
mov [esp-48], eax
mov [esp-52], ebx
mov [esp-56], edx
jmp instr_code
```

B's compiled code

B's Instrumented code
Arnold-Ryder [PLDI 01]: Full Duplication Profiling

Generate two copies of a method
- Execute “fast path” most of the time
- Jump to “slow path” occasionally to collect desired profile
- Adopted by IBM J9 VM; also used in “Bursty Tracing” [Hirzel&Chilimbi’01]
- Disadvantage: space/compile time
- Advantage: simple

Full-Duplication Framework

Checking Code

Duplicated Code

Method Entry

Checks

Check Placement
- Entry
- Backedges
Myth VI - Infrastructure requirements stifle innovation in this field

**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
    - [http://jikesrvm.sourceforge.net](http://jikesrvm.sourceforge.net)
  - ORP
    - [http://orp.sourceforge.net](http://orp.sourceforge.net)
  - Mono
    - [http://go-mono.com](http://go-mono.com)
  - < Insert your favorite infrastructure here >
But my open source VM doesn’t run “real” applications

- How do you expect me to do research on “real world software” if it doesn’t run on my VM?

  - **Answer 1:** Often it does
    - Jikes RVM has run Eclipse
      - A popular, open source IDE (Large, object-oriented Java app.)
      - Dozens of research papers written using Jikes RVM
        - None have measured performance impact on Eclipse
  
  - **Answer 2:** there are still things you can do (ala Mark Stoodley MRE’05)
    - Focus on the largest applications available, not smallest
    - Develop solutions that have potential to scale for real applications
      - Assume large “warm” code base, multiple threads, etc
    - Report metrics to demonstrate scalability
      - Example: for profiling technique
        - Report size of data collected per method
      - Program committees should reward this behavior
No shortage of research problems for virtual machines

- Higher-level optimizations
  - General purpose components, using tiny fraction of functionality
  - Higher-level programming models (e.g. J2EE, XML, Web Services, BPEL)
- Traditional optimizations, but for non-“toy” benchmarks
  - Selective optimization for programs with 30,000 methods
  - Inlining for call stack > 200 deep
- More aggressive use of speculation
  - Dynamic compiler look too much like traditional static compilers
- Stability of performance
  - Too many ad-hoc optimizations based on (poorly tuned) heuristics
  - React to phase shifts
- Optimizations for locality
  - New challenges and opportunities in managed runtimes
- Online interprocedural analysis
  - Mostly unexplored
  - Take a more global view of optimization
- How to exploit new hardware designs
  - Multicore, hardware performance monitors
- Resource-constrained devices (space, power …)
- Reducing complexity
Conclusions

- SE demands and processor frequency scaling issues require software optimization to deliver performance

- VMs are mainstream and are growing in importance
  - Get on board, or watch the train go by

- Commercial JVM implementations are on the cutting edge of research in adaptive optimization
  - Researchers not leading the field as much as they could
  - Whole program analysis of “Java” is not relevant to JVM teams

- Dynamic languages require dynamic optimization
  - Speculative software optimization is ripe for research

- How can we encourage VM awareness (for research and teaching) in universities?
Additional Information

- PLDI’04 Tutorial Slides
  Virtual Machine Learning, CGO’05 Keynote
  - Available on Michael Hind’s web page

- “A Survey of Adaptive Optimization in Virtual Machines”
  - by Arnold, Fink, Grove, Hind and Sweeney.
  - Proceedings of the IEEE, Feb 2005
  - Contains tons of citations

- 3-day Future of Virtual Execution Environments Workshop, Sept’04
  - 32 experts, hosted by IBM
  - Slides and video for most talk and discussion are available
  - Link off Michael Hind’s web page

- VEE’05 Conference, June 11-12, co-located with PLDI’05
  - Broad scope (includes language-level and OS-level VMs)
  - Keynotes from Jim Smith and Martin Nally
Appendix

Acknowledgements (ideas and inspiration)
- Michael Hind, Vas Bala, Bob Blainey, Perry Cheng, Stephen Fink, Matthew Arnold, Martin Hirzel, Nick Mitchell, Feng Qian, Gary Sevitzky, Jim Smith, Harini Srinivasan, Mark Stoodley, Peter Sweeney, Mark Wegman, Ben Zorn, ...