Virtual Machine Learning: Thinking Like a Computer Architect

Michael Hind
IBM T.J. Watson Research Center

What is this talk about?

- Virtual Machines?

What is this talk about?

- Virtual Machines?
  - YES!

What is this talk about?

- Virtual Machines?
  - YES!
- Machine Learning?

What is this talk about?

- Virtual Machines?
  - YES!
- Machine Learning?
  - NO!

Outline

- Why Performance Matters
- Mathematicians and Gamblers
- Are Today's VM Gambling Now?
- A Call for More Gambling
- Other Issues and 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.

  have helped enable the creation of large, sophisticated applications
- Resulting in modern languages with many benefits
  - Better abstractions & reduced programmer efforts
  - Better (static and dynamic) error detection
  - Significant reuse of libraries

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

How Have We Done?

- So far, so good …
  - Today's typical application on today's hardware runs as fast as 1970s typical application on 1970s typical hardware
  - Today's application is much more sophisticated
  - 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 development is still complex
  - PL/SE innovation will continue to occur
  - Trend towards more late binding, resulting in dynamic requirements
  - Will pose further performance challenges
- Processor speed advances not as great as in the past (x << X?)
- Computer architects providing multicore machines
  - Will require software to utilize these resources
  - Not clear if it will contribute more than in the past (y ? Y)
- Software implementation must pick up the slack (z > Z)
  - Because languages are becoming **more** dynamic, dynamic/speculative approaches are needed

Type Safe, OO, VM-implemented Languages Are Mainstream

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

Outline

- Why Performance Matters
- Mathematicians and Gamblers
- Are Today's VM Gambling Now?
- A Call for More Gambling
- Other Issues and Conclusions

Traditional Software Optimization

- Proves properties of the program by analyzing its structure
  - True for all executions
- Performed by compiler prior to execution

"Mathematicians"

optimization based on *proving* properties

Picking lottery numbers by analyzing the structure of balls

Hardware Optimization

- Speculates that the past will predict the future
- Occurs at runtime
  - No pre-execution analysis

Gamblers

optimization by *speculating* on runtime properties

Picking lottery numbers based on past results

Mathematicians vs Gamblers

- Pre-execution
- Proves properties for all executions

- Occurs at Runtime
- Speculates based on current execution

"Mathematicians"
Compiler Optimizers

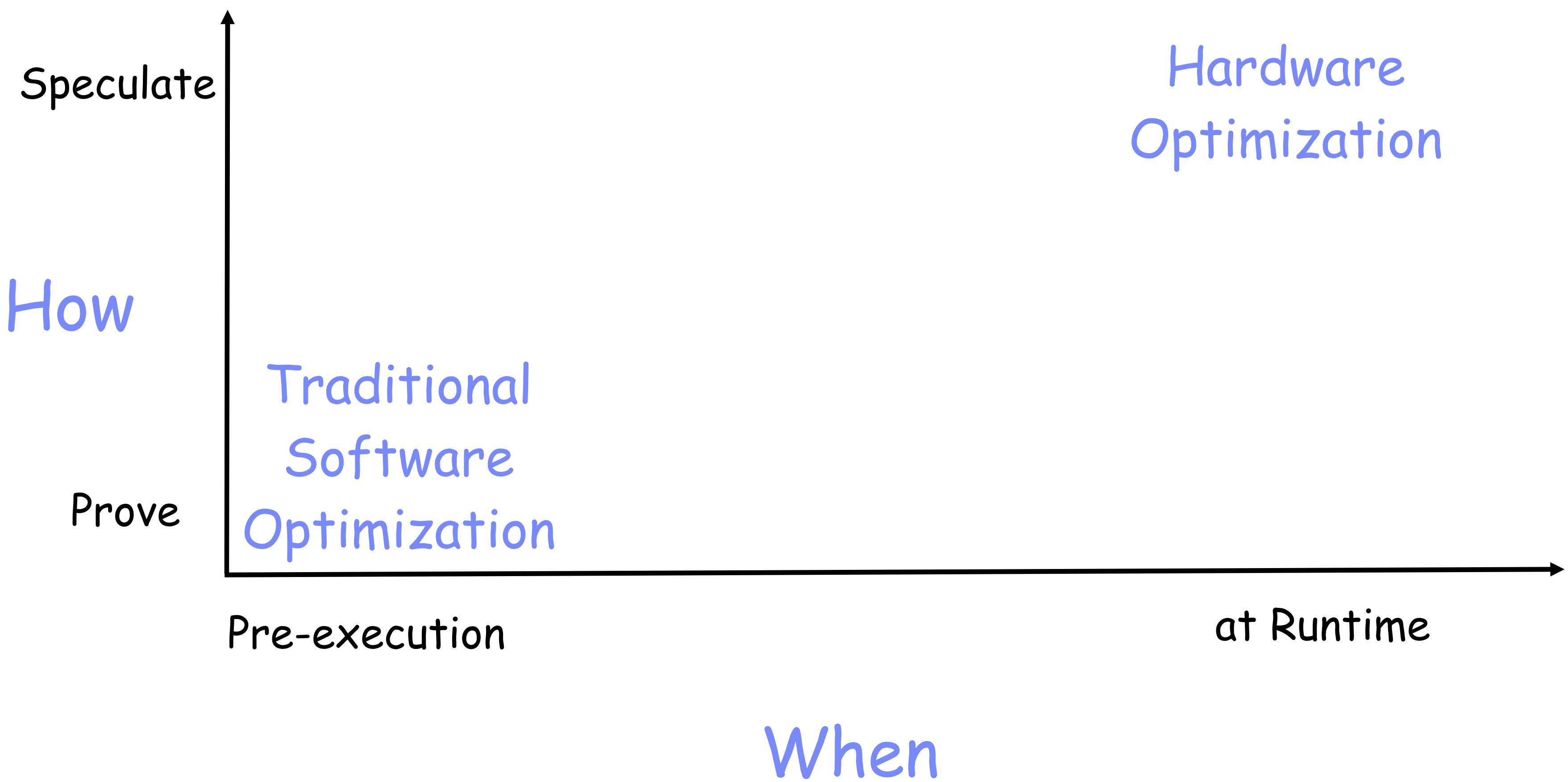
"Gamblers"
Hardware

What about a JIT?

Occurs at runtime

Prove or speculate?

Mathematicians vs Gamblers



Outline

- Why Performance Matters
- Mathematicians and Gamblers
- Are Today's VM Gambling Now?
- A Call for More Gambling
- Other Issues and Conclusions

How Do Today's VMs Achieve High Performance?

- Efficient Memory Management
  - Quick allocation performed in parallel
  - Garbage collection performed in parallel, mostly concurrent w/ app
- Efficient VM Services
  - Synchronization, method/interface dispatch, etc.
- Dynamic Compiler ("JIT")
  - . . .

What Is a JIT Compiler?

- Code generation component of a virtual machine
- Compilation is interspersed with program execution
- Program compiled incrementally; unit of compilation is a method
  - Compiler may never see the whole program
  - Must modify traditional notions of IPA (Interprocedural Analysis)

Similar to a Traditional Compiler?

- Build an IR, CFG, SSA, etc.
- Contains all traditional optimizations
  - SSA-based opts, graph coloring register allocation (HotSpot), etc.
- Heavy use of inlining
- Support multiple optimization levels

What's the difference?

- How the JIT is used!

- To achieve high performance ➔ need full suite of optimizations
- But overhead is too high
  - Compile time == runtime!
- Systems use selective optimization strategies
  - Assume methods are unimportant until profile says otherwise
  - Use interpreter or dumb compiler initially
  - Use JIT for the subset of important methods
  - eg. Self, HotSpot, IBM VMs, JRocket, ORP, Jikes RVM, etc.

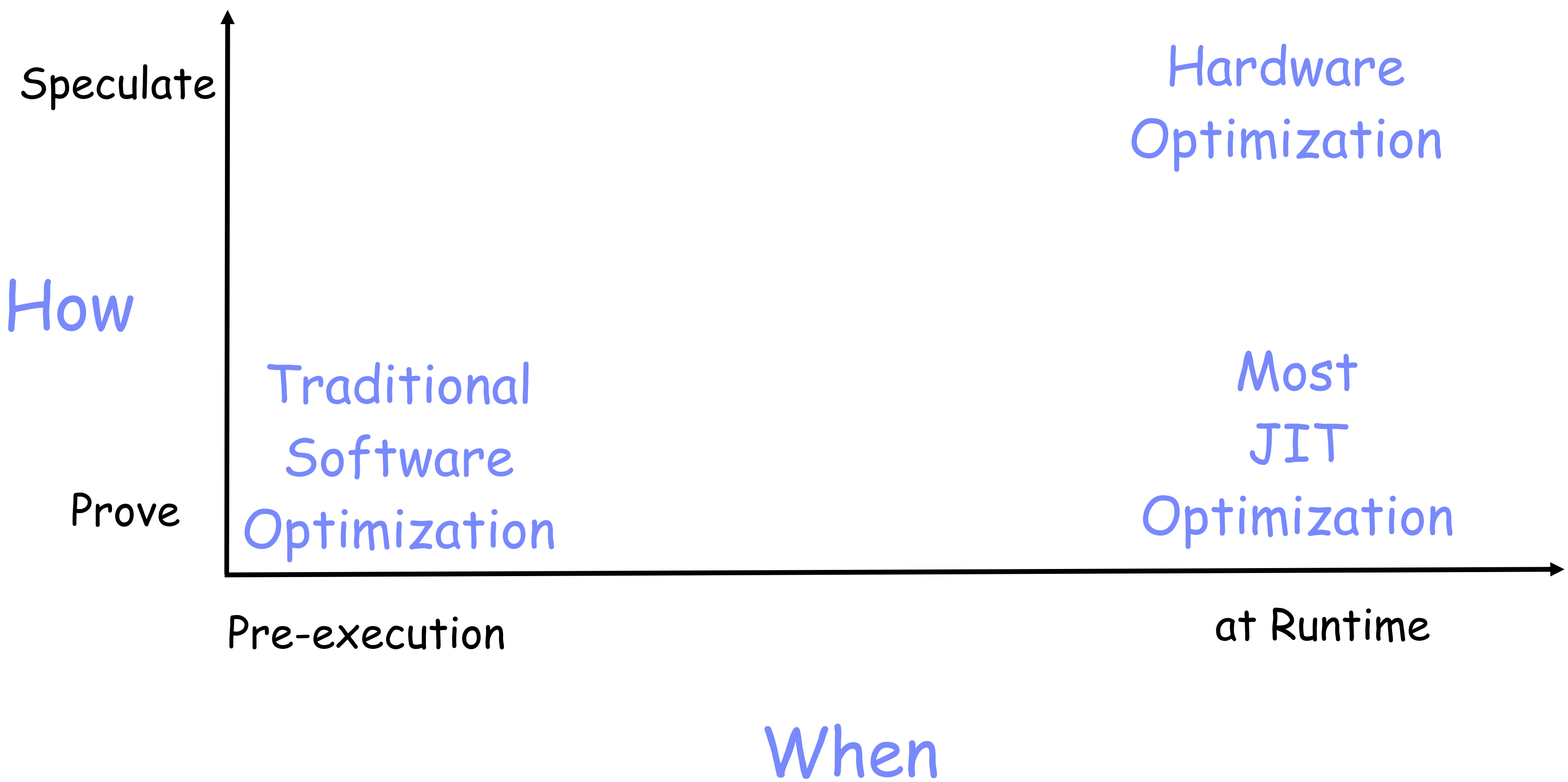
Profiling is being used to determine ***what*** to compile, i.e., to avoid the bad news of high overhead

But it is not all about avoiding bad news…

- Because compilation occurs at runtime, profile information can be used to guide heuristic-based optimizations
  - Similar to offline profile-guided optimization
  - Only requires 1 run!

- Performed in many systems
  - use branch profiles to infer basic block hotness
    - Input to code layout, register allocation, loop unrolling, etc.

Mathematicians vs Gamblers



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

JIT world does 2 & 3a, but not much of 3b

Outline

- Why Performance Matters
- Mathematicians and Gamblers
- Are Today's VM Gambling Now?
- A Call for More Gambling
- Other Issues and Conclusions

Speculative Approaches Are Common

- Hardware/OS
  - Caching, prefetching, scheduling, speculative HW, …
- Within a VM
  - Generational Garbage Collection
    - Assume objects die young until proven otherwise
  - Polymorphic Inline Caches for Virtual Method Invocation
    - Assume call target will be the same as last time
  - Synchronization
    - Assume a lock won't be contended
  - Hashing
    - Assume an object's hashcode won't be requested
  - Selective Optimization
    - Assume a method is not important

Why Aren't Speculative Strategies Used More in a JIT?

- Requires more infrastructure
  - Mechanism to gather profile to speculate on
  - Speculative strategy
  - Invalidation mechanism
    - Detect if speculation is wrong
    - Correct previous strategy
  - Introduces overhead and complexity
- Most of JIT community are "Mathematicians"
  - Dynamic compiler looks a lot like a static compiler
    - Dynamic compiler writers look like static compiler writers
  - Contrast this to dynamic binary optimizers
    - Similar problem, but different community
    - Much more gambling occurs!

Why Should VMs Be More Speculative?

- Applications are being composed at runtime, limiting program scope
- Modern languages are more dynamic, difficult to model
  - Pointers, virtual functions, dynamic class loading, scripting languages
- Unknown environment
  - Architecture implementation details, unknown libraries
- Can adapt to application's dynamic behavior
- Most of underlying system is already speculative
  - Opportunity for better synergy?
- A necessary step towards empirical optimization
- Significant performance gain may be possible

Ex. Suganuma et al. '02 says don't use any static heuristics for inlining, but instead rely on dynamic data

Example 1: Stack Allocation

Problem: OO languages encourage creation of many short-lived "local" objects

```
Complex c = new Complex(3.4, 2.1);
foo(c);
. . .
```

Optimization: allocate these objects on the stack rather than the heap (if legal)

- reduces pressure on garbage collector
- can improve data locality

Example 1: Approaches

- Ahead of time static optimization
  - Perform escape analysis of whole program
  - Proves which objects cannot escape their stack frame
  - Lots of papers, but doesn't work for dynamic language like Java
- Runtime static optimization
  - Perform local escape analysis on this method (and inlined methods) to prove it cannot escape
  - Conservative assumptions for other calls
- Speculative [Qian&Hendren'02]
  1. Assume the object will not escape
  2. Use write barriers to check to see if it does
  3. Adjust and learn from result in the future

Example 2: Interprocedural Analysis

Problem: OO languages encourage small methods

Solution:

- Construct a call graph that represents calling relation among methods
- Propagate method summary information along call graph
- Avoid pessimistic assumptions at call sites

Example 2: Approaches

- Ahead of time static optimization
  - Analyze whole program, building a call graph and propagate
  - Lots of papers, but doesn't work for dynamic language like Java
- Runtime static optimization [Hirzel et al'04]
  - Capture information as methods are compiled
  - Solve interprocedural problem when needed during execution
    - eg. at GC time
    - A form of incremental analysis
- Speculative [Qian&Hendren'04]
  1. Instrument calls to track actual targets, building dynamic call graph
  2. Provides speculative interprocedural information to optimizations
  3. May need to invalidate

Is Static Runtime Analysis a Bad Idea?

- NO!!!
- If something can be proven easily, then prove it
  - Ex. Loop invariant code motion
- However, a speculative approach provides other opportunities
- Interesting Hybrid
  - Combine static runtime with speculative [Hirzel et al'04]
    - Capture information as methods are compiled
    - But reflective code results are unpredictable
    - Track actual values used in reflective calls and use them as facts

Outline

- Why Performance Matters
- Mathematicians and Gamblers
- Are Today's VM Gambling Now?
- A Call for More Gambling
- Other Issues and Conclusions

Other VM Issues – Part 1

- Language interoperability
  - Applications in different managed languages?
    - Single VM vs co-operative VMs
  - Managed applications and native applications?
- How many levels of virtualization do we need?
  - Application server, OS, hypervisor, HW are all in the game
  - Are they even aware of each other?
- Security & Metadata
- Reliable, Predictable, Usable
  - Why do we have to specify a heap size?

Other VM Issues – Part 2

- HW Interactions
  - Do HW folks need us now?
  - Return of HW VM, or co-designed VMs
  - Better use of existing HW support in VMs
  - What about phase detection?
    - Hardware solutions are robust
    - Software optimization tend to ignore issue, but have larger scope
- Dealing with the implementation complexity
  - Layering, non-determinism, etc.
  - Can we speculate without adding complexity?

Conclusions

- VMs are mainstream and are growing in importance
  - Get on board, or watch the train go by
- SE demands and processor frequency scaling issues require software optimization to deliver performance
- Dynamic languages require dynamic optimization
- Current JIT strategies are not gambling enough
  - Speculative software optimization is ripe for research
  - But, we need to deal with the complexity!
- How can we encourage VM awareness in universities?

Appendix

Acknowledgements (ideas and inspiration)

- Matt Arnold, Vas Bala, Bob Blainey, Perry Cheng, Stephen Fink, Dave Grove, Martin Hirzel, Feng Qian, Jim Smith, Mark Stoodley, Peter Sweeney, Mark Wegman, Ben Zorn, . . .
- Future of VEE Workshop attendees
  - 2.5 days of slides and video available
- Brad Calder

Additional Resources (available on my web page)

- Survey paper on Adaptive Optimization in VMs, IEEE Proceedings, Feb'05 by Arnold, Fink, Grove, Hind, Sweeney
- 3+ hour tutorial on Dynamic Compilation and Adaptive Optimization

VEE'05 Conference, June 11-12, Chicago

