A Brief History of Just-In-Time

Presented by
Xuanran Zong and Jason Pazis

March 30, 2010

Introduction

- What is JIT compilation?
  - Any translation performed dynamically
  - After a program has started execution

Introduction

- What is JIT compilation?
  - Any translation performed dynamically
  - After a program has started execution
- Why do we care?
  - May offer advantages over static compilation and translation
  - Has received much attention in recent years
  - Many concepts have been reinvented

Advantages

- Advantages over compiled programs
  - Typically smaller in size
  - More portable
  - Access to run-time information

Advantages

- Advantages over compiled programs
  - Typically smaller in size
  - More portable
  - Access to run-time information
- Advantages over interpreted programs
  - Faster execution

History

- 1960-1969
  - Earliest published work
- 1970-1979
  - Prioritize space optimizations
  - Optimization of "hot spots"
- 1980-1989
  - JIT similar to memory management
  - Aggressive JIT customization
- 1990-2000
  - Spreading out compilation time
  - Slim binaries
  - Staged compilation

History

- 2000-today
  - Huge investment of time and money into JIT
  - Concurrent development of many approaches
  - Reinvention of many concepts

Space optimizations

- Mixed code
  - Compile “hot spots” only
  - Fine grained mixture implied

Space optimizations

- Mixed code
  - Compile “hot spots” only
  - Fine grained mixture implied
- Caveats
  - Both a compiler and interpreter need to be in memory
  - Both a compiler and interpreter need to be maintained

Space optimizations

- Throw-away code
  - Compile a block
  - Execute it
  - Discard the code

Space optimizations

- Throw-away code
  - Compile a block
  - Execute it
  - Discard the code
- Alternatively
  - Keep a fixed-size cache of program code

Space optimizations

- Throw-away code
    - Compile a block
    - Execute it
    - Discard the code
- Alternatively
    - Keep a fixed-size cache of program code
- Deferred compilation of uncommon cases
    - Some cases may never be compiled during typical execution

Dynamic and adaptive optimization

- Collect run-time information
  - Utilize runtime type information
  - Maintain execution counters
  - Perform optimizations in order of increasing complexity
  - Concentrate effort on hot-spots

Dynamic and adaptive optimization

- Collect run-time information
  - Utilize runtime type information
  - Maintain execution counters
  - Perform optimizations in order of increasing complexity
  - Concentrate effort on hot-spots
- Considerations
  - “What” to optimize is more important than “when”
  - The code that should be optimized may not be the one that triggered the optimization
  - Inlining may be the answer to frequently called short methods
  - Cache coherency issues may arise
  - Implicit or explicit invocation of JIT compiler (Erlang)

Dynamic and adaptive optimization

- Alternatively
  - Start by profiling
  - Amortize profiling cost over program execution

Continuous run-time optimization

- Input may vary over time
  - Similar in spirit to cache strategies
  - Optimize based on the most recent input patterns
  - Dynamically reorder code

Continuous run-time optimization

- Input may vary over time
  - Similar in spirit to cache strategies
  - Optimize based on the most recent input patterns
  - Dynamically reorder code
- Possible extensions
  - Dynamically optimize based on available resources
  - The resources available may change
  - Resource utilization may change due to contention

Customization

- Customize frequently executed methods
  - Many classical compiler techniques apply
  - Constant propagation/folding
  - Dead code elimination

Customization

- Customize frequently executed methods
  - Many classical compiler techniques apply
  - Constant propagation/folding
  - Dead code elimination
- Specialize interpreter's instructions
  - Reduce the overhead of instruction dispatch
  - Yield opportunities for macro opcode optimization
  - The speedup obtained is significant
  - Does not compete with compilation
  - May not be as fruitful as other approaches

Improving the user experience

- Avoid large compilation spikes
  - Large periods of unresponsiveness degrade user experience
  - Favor response time over total execution time
  - Perform optimizations in stages

Improving the user experience

- Avoid large compilation spikes
  - Large periods of unresponsiveness degrade user experience
  - Favor response time over total execution time
  - Perform optimizations in stages
- Spread out compilation time
  - Two or more closely followed compilation pauses
  - Can appear as one large pause

Improving the user experience

- Avoid large compilation spikes
  - Large periods of unresponsiveness degrade user experience
  - Favor response time over total execution time
  - Perform optimizations in stages
- Spread out compilation time
  - Two or more closely followed compilation pauses
  - Can appear as one large pause
- Deferred compilation of uncommon cases
  - Compile only the current execution path
  - Set up stubs for non-compiled cases
  - Responsiveness can improve significantly for large case blocks
- Fast code generation

Machine independent code representation

- Slim binary, Java, etc.
- Same executable vs heterogeneous substrate computing environment
- Slim binary: a high-level, machine independent program module
- Generate executable on-the-fly when loaded
- Generate module at once is superior to method-at-a-time

Simulators

- Running executable code for one architecture on another
  - Highly specialized with respect to source and target

Simulators

- Running executable code for one architecture on another
  - Highly specialized with respect to source and target
- Categories
  - Interpreters: First generation
  - Dynamically translated *instructions*: Second generation
  - Dynamically translated *blocks*: Third generation
    - Block-at-a-time or page-at-a-time
  - Dynamically translated *paths*: Fourth generation

Fourth generation

- Predominant in recent literature

Fourth generation

- Predominant in recent literature
- Common techniques
  - Profiled execution
  - Hot path detection: counter, code structure, sample PC
  - Code generation, optimization and verification
  - “Bail-out” mechanism

Fourth generation

- Predominant in recent literature
- Common techniques
  - Profiled execution
  - Hot path detection: counter, code structure, sample PC
  - Code generation, optimization and verification
  - "Bail-out" mechanism
- Recurring themes
  - Binary to binary translation
  - Legacy to VLIW code translation
  - Target architecture provide extra resource
  - Can these scale?

Classification of JIT systems

- Invocation
  - Implicit: transparent to the user
  - Explicit: Allows for better control
  - Currently implicit invocation systems dominate

Classification of JIT systems

- Invocation
  - Implicit: transparent to the user
  - Explicit: Allows for better control
  - Currently implicit invocation systems dominate
- Executability
  - Monoexecutable
  - Polyexecutable

Classification of JIT systems

- Concurrency
  - Concurrent execution and compilation
  - Execution stalls in order to compile
  - Concurrent systems are becoming more important

Classification of JIT systems

- Concurrency
  - Concurrent execution and compilation
  - Execution stalls in order to compile
  - Concurrent systems are becoming more important
- Hard real time systems
  - Little research in this area
  - Usually developed using worst case analysis
  - Implicit compiler invocation incompatible

Issues faced by JIT tools

- Binary code generation
  - Rife with opportunities for error
  - Lots of bookkeeping tasks

Issues faced by JIT tools

- Binary code generation
  - Rife with opportunities for error
  - Lots of bookkeeping tasks
- Cache coherence
  - Cache and memory can become out of sync
  - More complicated on shared memory multiprocessors

Issues faced by JIT tools

- Binary code generation
  - Rife with opportunities for error
  - Lots of bookkeeping tasks
- Cache coherence
  - Cache and memory can become out of sync
  - More complicated on shared memory multiprocessors
- Execution
  - Restrictions on where executable code can reside
  - Restrictions on which parts of memory can be edited

Java

- Static compilation to bytecode
- JIT compilation from bytecode to machine code
- Originally only interpreted
    - Surprisingly slow
- Different implementations have surfaced
    - Stack based, register based
    - Some skip the bytecode phase altogether
- Was the driving force for much research in JIT
- Other languages are now targeting the JVM

Java

- Other JIT compilation approaches
  - Compile-only strategy
  - Translate byte code into Self code to leverage existing optimization
  - Code annotation to facilitate code optimization prior to run-time
  - Continuous compilation for Java

Conclusion

- JIT compilation is an old technique
- It has received much attention in recent years
- It can result in smaller footprint than compiled code
- It has the potential to achieve
  - Better performance than interpreted code
  - Or even compiled code
- It can provide platform independence

Thank you for your attention

- Thank you for your attention
- Questions?