“Strictly speaking, JIT compilation systems [...] are completely unnecessary.” -- Aycock

JITs & VMs common today, however!
System that performs (nontrivial) code
translation or compilation after starting
execution.
Program Representations (Rau)

- High-Level Representation (HLR)
  - C++/Java
- Directly Interpretable Representation (DIR)
  - Bytecode
- Directly Executable Representation (DER)
  - x86 binary

A VM is software which executes DIR code.
Benefits of JITs & VMs

- Code portability
  - Compile once, run everywhere.
- Smaller programs
  - You can compile what matters and interpret the rest.
- Runtime information for optimization
  - Adaptive optimizations based on workload.
- Faster than vanilla interpretation
Tradeoffs

- Slower:
  - Interpreted source code
  - Interpreted virtual machine code
- Faster:
  - Native code

- Smaller:
  - Interpreted source code
- Larger:
  - Native code
Quick History

- 1960s: John McCarthy & Lisp
  - Memory management in VMs
  - Early precursor to dynamic loading
  - McCarthy suggests process where compilation is fast enough that results do not need to be saved
- 1970s: Hansen’s Adaptive Fortran
  - Thesis work focused on answering when, how, and what code should be optimized in a JIT.
Quick History

● 1980s: Smalltalk and Self
  ○ Full fledged JITs demonstrated that software VMs were viable.
  ○ Polymorphic caches, on-stack replacement, dynamic deoptimization, selective compilation, type prediction and splitting, profile-directed inlining.

● 1990s: Java and the JVM
  ○ First real commercial penetration
  ○ Sparked a lot of work looking at developing and improving JIT techniques.
Selective Optimization

Policy for when to use the runtime compiler.

- “JIT-only” strategy
  - Compile all code when needed.
- Selective Optimization
  - Profiling Mechanism: identify candidates for optimization.
  - Decision-Making Component: Determine which optimizations to apply.
  - Dynamic Optimizing Compiler: Apply optimizations
Profiling for Selective Optimization

2 Standard Approaches

● Counters
  ○ Increment each time you pass through a “code unit” (e.g. methods).

● Sampling
  ○ Periodically interrupt and check where you are in the code.

Pick a threshold at which to optimize.
Selecting for Selective Optimization

Multiple levels of optimization.
- Each level, recompile with additional optimization.

Sometimes paired with inlining decisions
- SELF-93, HotSpot Server VM

Careful to ensure that pauses for (re) compilation aren’t prohibitive.
On-Stack Replacement

Need a way to transfer from interpreter to compiled code in run time.

Replace interpreter stack frame with the newly compiled code’s stack frame.
Deferred and Partial Compilation

Not all paths in a method will be common.

Some JITs use *deferred compilation* for uncommon code branches.

Trampoline/trap code to run the interpreter (or compiler) for branch when executed.
Supporting Debugging and Introspection

Dynamic deoptimization to support stepping through with a debugger.

Use OSR to switch to unoptimized code at breakpoint.
Feedback Directed Optimization

Motivation:

- Overcomes limitations of static optimization.
- Allows the system to change or revert optimizations.

Challenges

- Compensating for overhead for collecting information.
- Accounting for possibly partial profile information.
Profiling for FDO

- Runtime Service Monitoring
  - Information from services such as memory management.
- Hardware Performance Monitors
  - Not common.
- Sampling
- Program Instrumentation
- Combined Sampling and Instrumentation
Some optimizations assume profile stability.

- Oberon VM (Kistler) used a periodically captured vector of event counts.
  - Vector became too different from previous measurements => new program phase.
- Dynamo binary translator detected phases by looking for increases of new optimization events.
FDO Code Generation

Inlining:

- In traditional compilers, use KNAPSACK.
  - JITs don’t use this since it requires global reasoning.
- In JITs:
  - Ad hoc heuristics
  - Inlining Trials (Dean and Chambers)
FDO Code Generation

Code layout:

- Goal is to improve instruction locality by laying out common paths to be contiguous.
- Most systems use a variant of approach detailed by Pettis and Hansen
FDO Code Generation

Instruction Scheduling:

- **Goal:** Order instructions to best avoid bubbles in processor pipeline.
- **Difficult since it requires targeting specific hardware.**
FDO Code Generation

Multiversioning:

● Create multiple versions of the code and pick the best one

● Most commonly associated with speculating for dynamic dispatch
  ○ Use lightweight inline guards to see if the speculation was wrong, correct with OSR.
JITs and VMs have been around for a long time.

Provides a way to make code mobile and perform dynamic optimizations.