Dynamic compilation

Beyond interpretation

- Recap: interpreters interpret one "thing" (node, bytecode) at a time, and (typically) don't remember anything (or, not much)
- Compilation: consider many AST nodes/bytecodes together, and emit equivalent machine code once, which can be reused many times
  - The compiler can analyze a larger region of code, which will typically reveal more optimization opportunities.
  - Reason to introduce JIT The compilation process can be expensive, but the goal is to amortize the cost by executing the fast machine code many times
  - Compiling code which is executed once, or a few times, is generally not a win

Why *dynamic* compilation?

- Traditionally, compilation is *static* (ie done by the developer) and results in a *machine code binary* which is distributed to end users

- Why not do this all the time?

Why *dynamic* compilation?

- ✓ Binaries are not portable
- ✓ Binaries are hard to verify for security properties type info lost, control flow changes.
- ✓ Might not know enough ahead-of-time to generate good code may use some run-time info
- Binaries are big:
  - ✓ compile everything, even though much code may never be executed
  - ✓ binary ISA encodings are often not compact (compared to bytecode)

Other advantages of dynamic compilation

- The program is running: data values can be inspected, actual execution observed.
- The specific model of CPU is known; can select model-specific instructions; size of memory, structure of memory hierarchy are known.

Static compilation must focus on minimal framework.
But dynamic compilation can acquire concrete model.

Compiler design choices
1. Unit size

1. How much to compile in one go (the *unit* of compilation):

    - A basic block — too small (see selective inlining)
    - A method
    - Many methods (via inlining)
    - A trace (later)

Compiler design choices 2. When

1. **Ahead of time**, i.e. well before the first execution (but after bytecode has been distributed)
   - E.g., in Java™, upon completion of class loading
   - Another possibility is at install time
2. **"Just in time"**
   - Compilation is triggered by the first execution, which is delayed until compilation is complete
3. After some number of interpretations*
   - **One**
     - E.g., after resolution has been performed in Java
   - **Many**
     - To gather execution information (detect hot loops, gather type profiles)

Compiler design choices 3. Level of optimization

1. ✓ Minimal: macro expansion of bytecode action routines

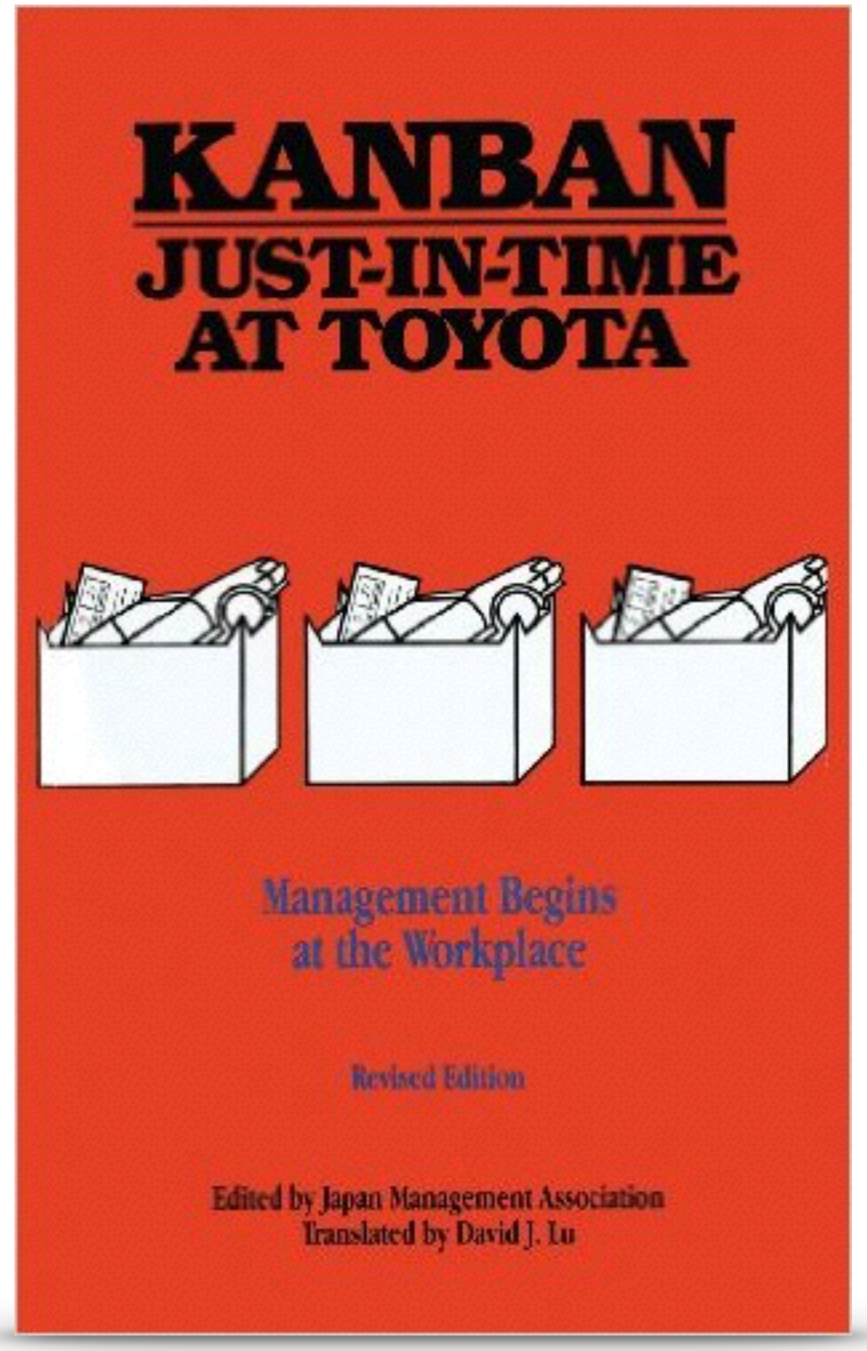
manually register allocation. like stack + one register.

2. ✓ Simple: (register management "by hand"), peephole optimization

3. Advanced — the sky's the limit. More later.

Just-In-Time Compilation

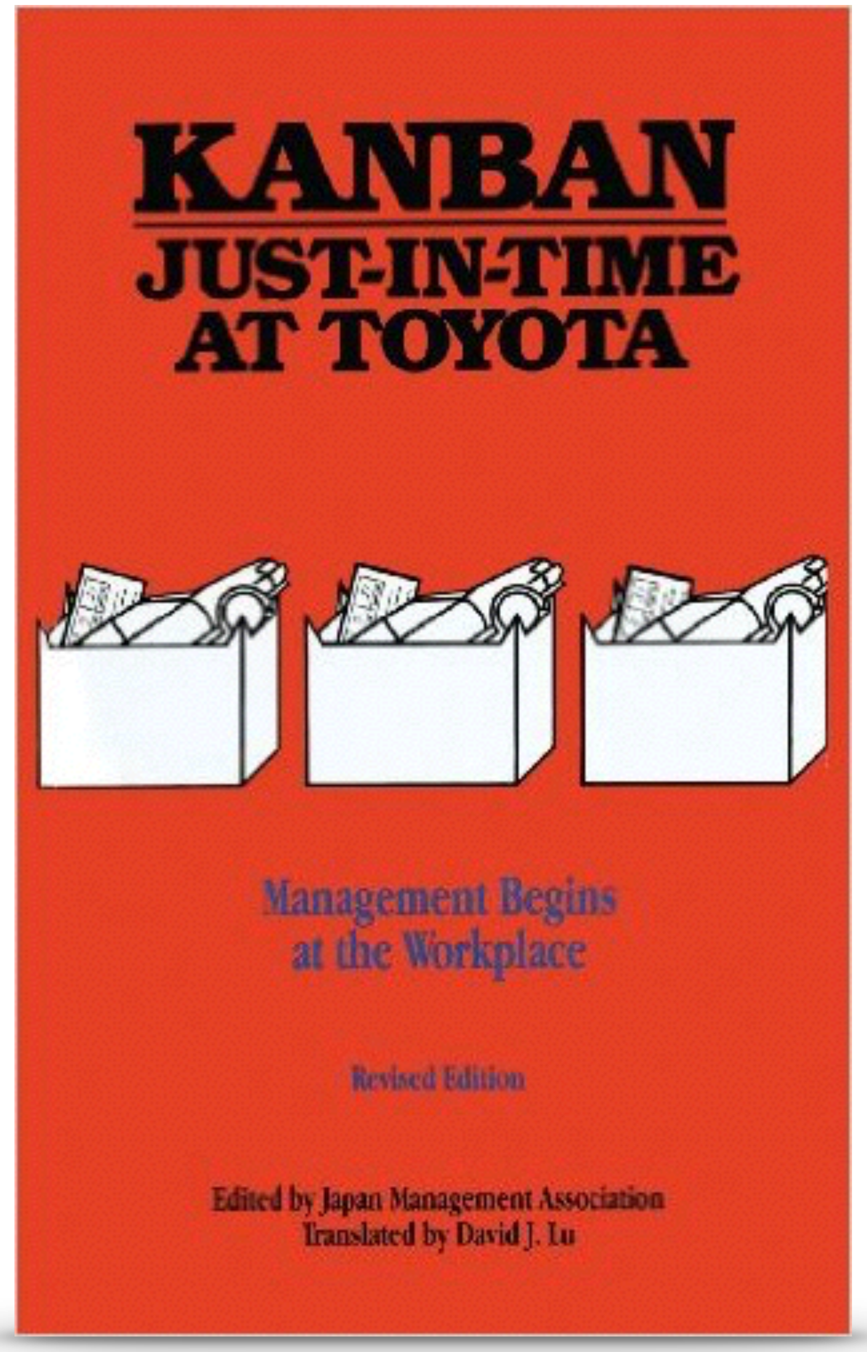
- Term originates from just-in-time manufacturing (aka *kanban* method)
- Appeared around the time of Java's uptake
- Terrible misnomer (should be "just too late")
- Universally misapplied (e.g., to dynamic compilation *after* first execution)
- "JIT" is **not a noun**


KANBAN
JUST-IN-TIME AT TOYOTA
Management Begins at the Workplace
Revised Edition
Edited by Japan Management Association
Translated by David J. Lu

✓Just-In-Time Compilation

- Term originates from just-in-time manufacturing (aka *kanban* method)
- Appeared around the time of Java's uptake
- Terrible misnomer (should be "just too late")
- Universally misapplied (e.g., to dynamic compilation *after* first execution)
- "JIT" is **not a noun**

Suggestion: "jit" as a new noun, meaning dynamic compiler.



KANBAN
JUST-IN-TIME
AT TOYOTA

Management Begins
at the Workplace

Revised Edition

Edited by Japan Management Association
Translated by David J. Lu

A JIT compiler eliminates interpreter dispatch overhead

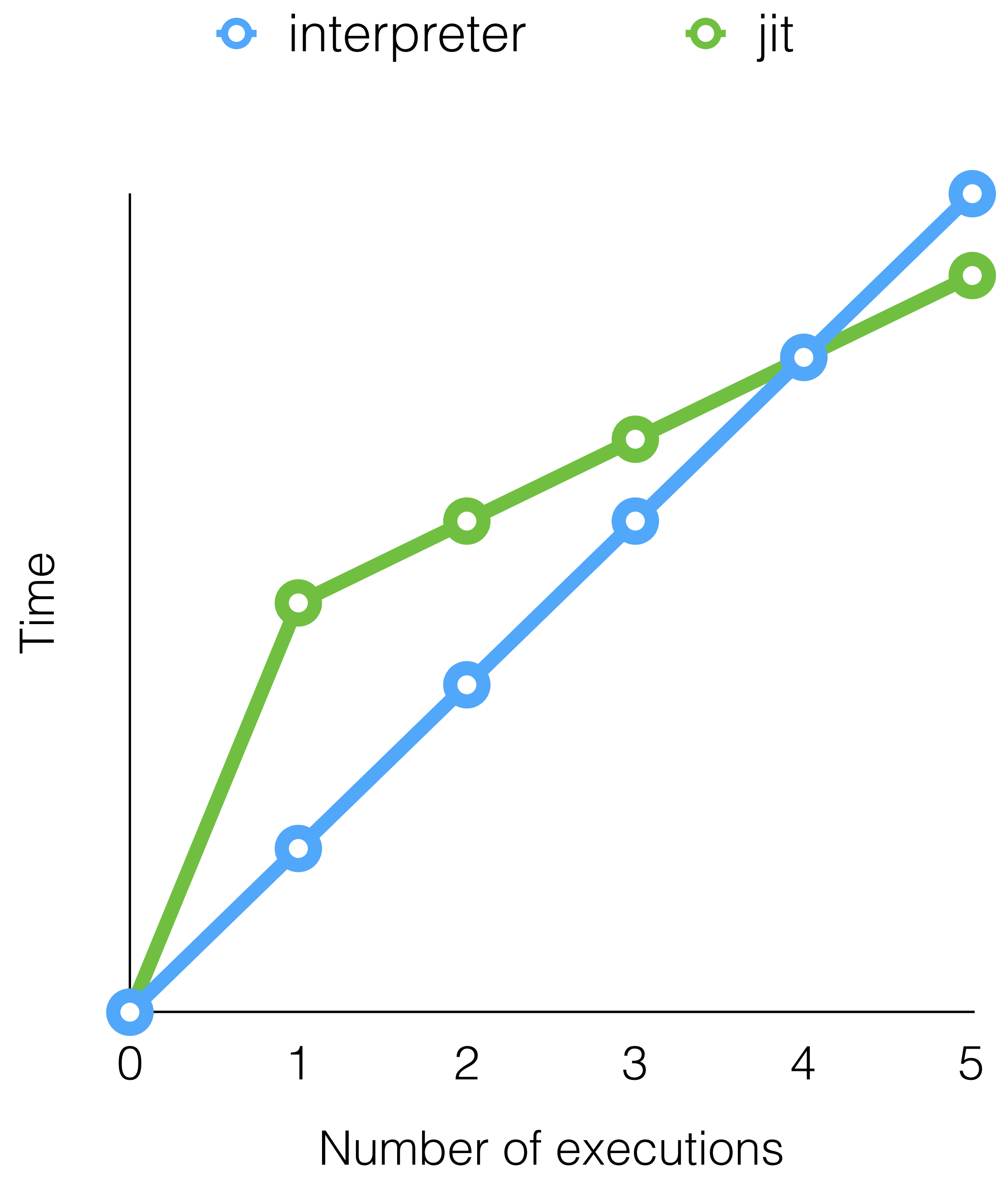
- Simple elimination of interpreter overhead leads to a significant speedup. This is because we perform bytecode decoding only once, and the generated code does not have the overhead of the interpreter loop (or other dispatch for more advanced forms of interpretation).

- The resulting code can be no more complex than the machine code of each interpreter action corresponding to the sequence of bytecodes being compiled, with simple constant folding; little more than macro expansion.

A simple model of “JIT compilation”

Three defining parameters:

1. Speed of interpretation
2. Cost of compilation
3. Speed of compiled code



Simple speed metrics

In measuring speed, there are countless correlated variables, but a simplifying approach is to establish a **benchmark suite**, and then measure relative to the **average bytecode** .

- Measure CPU time for the whole suite, subtract out GC, native code and other unrelated execution, and divide by the number of executed byte codes.

- Commonly, use clock cycles as a proxy for time.

Simple speed metrics

- The resulting measures are (units: cycles per bytecode):
  - Interpretation time, t_I
  - Compilation time, t_C
  - Execution time of compiled code, t_E
- And even simpler are two ratios (both should be >1):
  - Compiler : interpreter, $\tau = t_C / t_I$ (mnemonic: translate)

    $\tau \rightarrow$ compilation time larger than interpretation time.
  - Interpreter : compiled code, $\rho = t_I / t_E$ ("run")

    $\rho \rightarrow$ interpretation time larger than execution time of compiled machine code.

The break-even point

After β times' execution of compiled code, the total time is the same as interpreter β times.

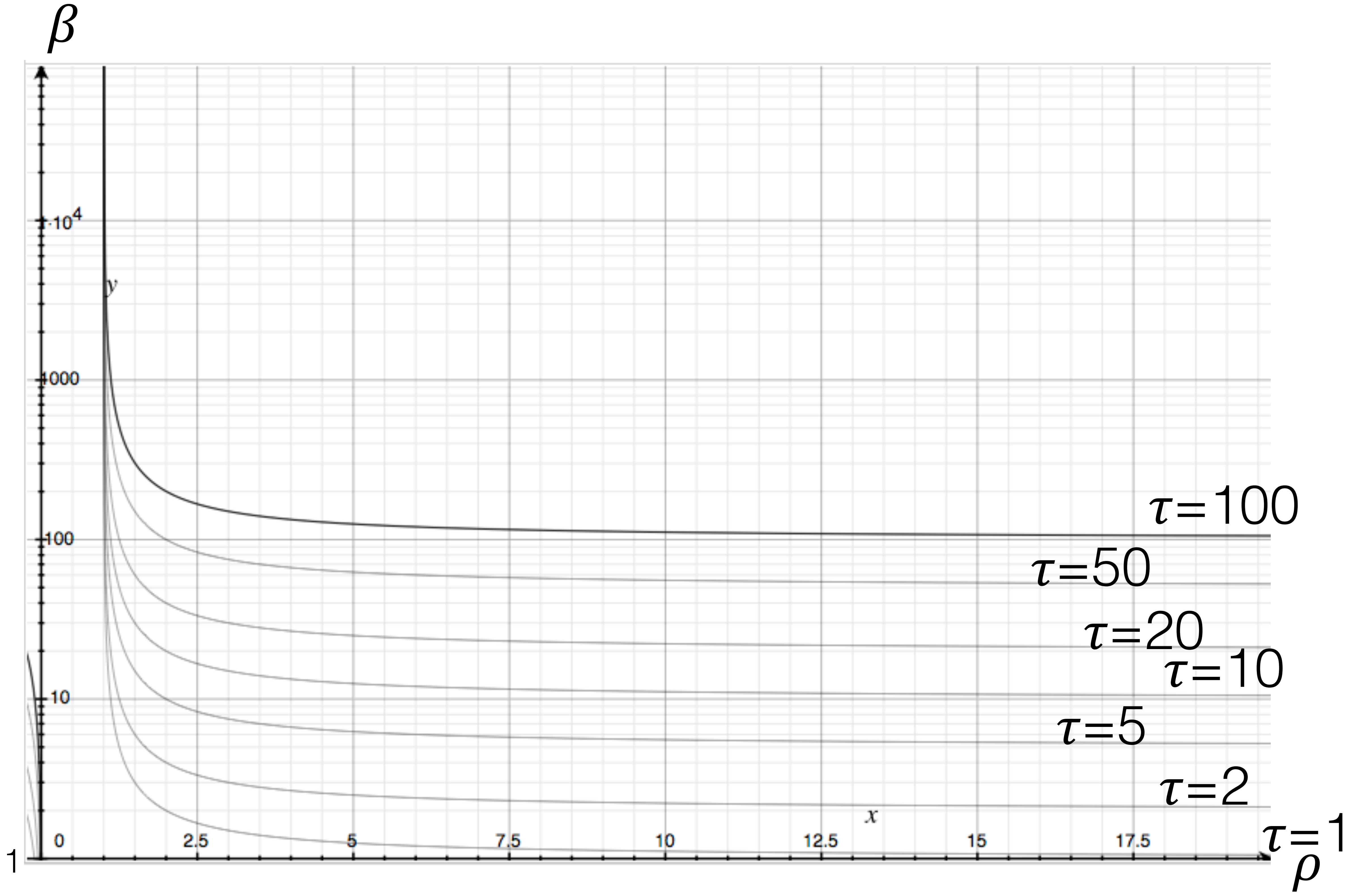
Simple math yields that the break-even number of executions of compiled code is

$$\beta = \frac{t_C}{t_I - t_E} = \frac{\rho\tau}{\rho - 1}$$

Example: $t_E=1$, $t_I=2$, $t_C=10$, $\tau=5$, $\rho=2$, $\beta=10$

Formula: $\beta \cdot t_I = \beta \cdot t_E + t_C \Rightarrow \beta = \frac{t_C}{t_I - t_E} = \frac{t_C/t_E}{t_I/t_E - 1} = \frac{\tau \cdot \rho}{\rho - 1}$

log β vs. ρ for τ=1, 2, 5, 10, 20, 50, 100

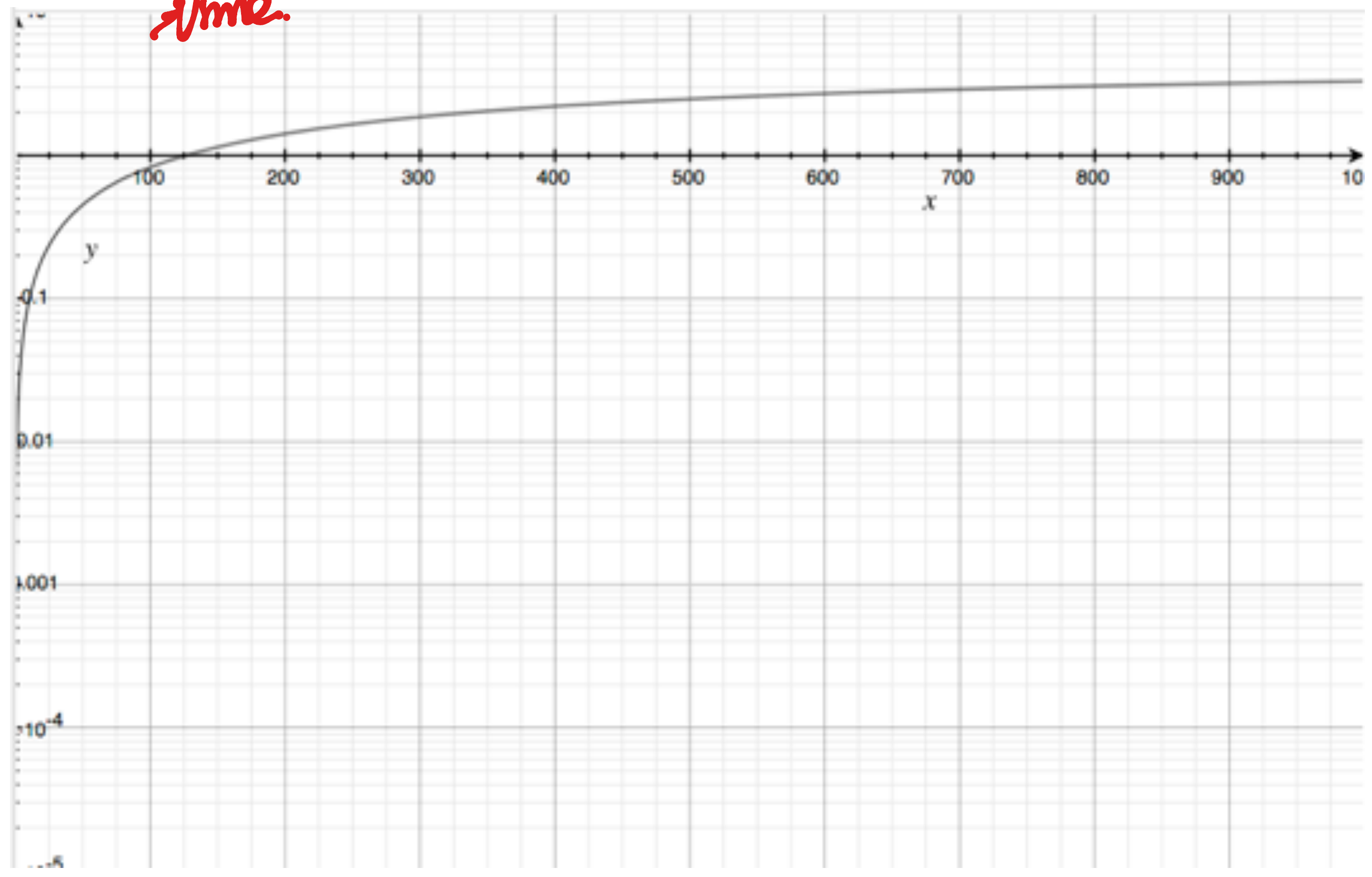


Speedup

If a bytecode is executed *n* times, then the speedup is given by $\dfrac{nt_I}{\boxed{t_C + nt_E}}$ or $\dfrac{n\rho}{n+\rho\tau}$.

total execution time $\leftarrow$

1



log speedup vs. *n*
for ρ=5, τ=100
β=125

Examples

```
#define o(OP) (OP<<5)
#define L5(OP) (OP&0x1F)
#define H3(OP) (OP&0xE0)


while (1) {
  switch (H3(*pc)) {
  case o(LIT)  : *++sp= ((((pc[1]<<8)+pc[2])<<8)+pc[3])<<8)+pc[4];
                 pc += 5;
                 continue;
  case o(ADDop): sp[-1] += sp[0]; --sp; ++pc; continue;
  case o(SUBop): sp[-1] -= sp[0]; --sp; ++pc; continue;
  case o(MULop): sp[-1] *= sp[0]; --sp; ++pc; continue;
  case o(DIVop): sp[-1] /= sp[0]; --sp; ++pc; continue;
  case o(GET)  : *++sp= vars[L5(*pc)];  ++pc; continue;
  case o(PUT)  : vars[L5(*pc)]= *sp--;  ++pc; continue;
  case o(END)  : goto out;
  }
}
out:
```

Compiling a LIT

Interpreter:

```
case o(LIT): *++sp= ((((pc[1]<<8)+pc[2])<<8)+pc[3])<<8)+pc[4];
             pc += 5;
             continue;
```

Compiler:

```
case o(LIT):
  lit= ((((pc[1]<<8)+pc[2])<<8)+pc[3])<<8)+pc[4];
  emitPushLiteral(lit);
  pc += 5;
  continue;
```

RISC (SPARC™) assembly code to emit:

```
    add Rsp, 4, Rsp
followed by
    ldw Rlit[#lit], Rtemp ; use a literal frame per method
    stw Rtemp, [Rsp]
or
    sethi lit-hi-part, Rtemp
    or    lit-lo-part, Rtemp
    stw   Rtemp, [Rsp]
or
    stw lit, [Rsp] ; if lit can be immediate
```

Compiling an ADD

Interpreter:

```
case o(ADDop): sp[-1] += sp[0]; --sp; ++pc; continue;
```

Compiler:

```
case o(ADDop): emitAdd(); ++pc; continue;
```

```
emit:
   ldw [Rsp], Rtemp
   sub Rsp, 4, Rsp
   ldw [Rsp], Rtemp2
   add Rtemp, Rtemp2, Rtemp2
   stw Rtemp2, [Rsp]
```

Compiling a GET

```
Interpreter:

case o(GET): *++sp= vars[L5(*pc)];  ++pc; continue;


Compiler:

case o(GET): emitGet(&vars[L5(*pc)]);  ++pc; continue;


emit:
   ldw [varLoc], Rtemp
   add Rsp, 4, Rsp
   stw Rtemp,[Rsp]
```

Compiling a PUT

```
Interpreter:

case o(PUT): vars[L5(*pc)]= *sp--;  ++pc; continue;


Compiler:

case o(PUT): emitPut(&vars[L5(*pc)]);  ++pc; continue;


emit:
   ldw [Rsp], Rtemp
   sub Rsp, 4, Rsp
   stw Rtemp,[varLoc]
```

Calls into the runtime

- At any point in the compiled code, the compiler can transition into the static code of the VM to provide runtime support — you don’t have to emit code for *everything*.

- Example: generate code for the fast (successful) path of bump allocation, but jump into the runtime for the slow path.

Managing code

- Since you are generating code at run-time, you have to:
    - Find some place to put it
    - Be able to locate it when dispatching/linking
    - (Optional, but recommended) Discard code if you run out of space

Code cache

- These are the functions of a *code cache*.
- The compiler will emit code into a large buffer; in general you cannot know the emitted code size in advance of compilation.
- Once completed, the code is installed (by copying) into the code cache, and linked from the dispatch structure(s) [e.g., method lookup tables].
    - Copying — simple relocation — hence best to make calls into the runtime using absolute addresses.
- I call a compiled (native) method (or function) an *nmethod*.

Flushing code

- In some languages, GC can lead to code being unreachable (e.g., Java class unloading). It's good to free the nmethods when this happens.
- Also, if the code cache is full, one or more old nmethods will have to be flushed to make room for a new one.
    - De-link old from dispatch tables, install link to trigger lookup/compilation.
- What if the nmethod is currently active? (i.e., there is an activation record for this nmethod on a/the stack)

Finding active nmethods

- Either:
    - Increment an activation count on entry to the method, and decrement on exit (and thread termination, and exceptions, and …)
- Or:
    - Scan the stacks of all threads looking for activation records

Stack scanning

- The calling convention must allow for the linear scan of the stack of a suspended thread:
  - Access saved registers; extract SP, FP
  - Walk down the chain of saved frame pointers; test if the return address of a frame is within the nmethod under scrutiny.
- *Highly* machine- and calling-convention dependent.
- The OS can (and sometimes does) make this impossible (e.g., by saving register state in kernel space).

Active nmethods: flush or not?

Having found that an nmethod is active, we can:

- Choose to leave it alone, and look elsewhere
- Discard it, and rewrite the activation record so that when it is resumed, it resumes in the interpreter (or triggers another compilation), at the right place, with the right state — a technique known as *dynamic deoptimization*.

We'll look at the latter process in a future lecture.

Code cache: compact or not?

- Nmethods are of varying size. Hence, turnover in the code cache can lead to fragmentation.

- Do we compact the code cache or not?

  - Yes: extra complexity (suspend all threads, relocate nmethods, update all dispatch links, return addresses) — Self VM

  - No: flush code until we get a big enough hole for our new nmethod — HotSpot^{TM} JVM.

Nmethods in the heap?

- Another option is to make nmethods look like objects and put them in the heap.
- Can use the same GC and compaction, with nmethod-specific scanning/moving and unified heap management.
- Cons: code is usually *very* long-lived, compared to data objects.
- A tiny bug may let the application write data into the code.

Finding GC roots on the stack, in registers and in code

- Finding all the pointers in the presence of a *jit* and a moving collector presents new challenges:

    - Pointers may be in registers, in stack locations for the activations of nmethods, and even embedded in nmethods as literals (split across multiple instructions).

    - Worse, a register (or stack spill slot) may contain a pointer in the middle of (de)construction — being tagged or de-tagged, or undergoing address arithmetic.

Roots in registers (a) call sites

- All activations in a thread, except the top one, are at a call site.

- For each call site in an nmethod, the compiler can emit a *register map* that indicates which registers are live (if pointers can be untagged, this must be a map of live pointers).

- Example: When the Self compilers emit calls, after each call is a register map word: each bit represents a register, with the bit set if the register is live and has an oop. The return sequence skips over the map. A simple stack scan will locate all the register maps.

Roots in registers
(b) the top frame

In contrast, the top frame can be suspended at any instruction. Maintaining a register map for every instruction is prohibitive in space cost.

Some approaches from the literature:

- Ensure a suspended thread is at a *GC safe point*, for which a register map is available.

- Maintain register maps at various sites, and use *abstract interpretation of the machine code* to derive the map at any instruction in between.

- A compiler *replay*, requesting the register map(s), could also be used, but in general is probably too expensive.

Simple GC safe points

- The simplest approach is perhaps to insert GC safe points at nmethod entry and backward branches.
  - Ensures self-suspension in bounded time.
- At the safe point, a global flag is polled to see if the thread should suspend itself.
- When a GC is about to start, all threads are advanced until they suspend themselves at GC safe points (or are suspended in OS calls). Each thread could even scan its own stack before suspending.

Roots on the stack

- Similar to a register map, each call site needs to indicate which parts of the stack frame contain live oops or pointers. If tagged, then the bounds of a stack region usually suffice.
- These can be in a side structure (eg, a list of descriptors after the code of the nmethod), or
- A pointer to the structure can be placed after the call but before the following code.

Roots in code

- If the language admits literal object references, then the compiler may emit them into the code stream. On most ISAs, this involves piecemeal assembly from sub-bitfields.

- In this case, the compiler has to emit a side table identifying the locations of these references, and must be able to discover them as roots (and rewrite them should the object move).

Exercise

Modify your Feeny VM to use a JIT compiler in place of the interpreter.

Measure the changes in performance.

Compiler topics to come:

- Optimizations: assorted dispatch techniques; customization, inlining
- Feedback-driven optimization
- Speculative compilation techniques
- Dynamic deoptimization
- Tiered compilation