Advanced interpretation techniques

Faster interpretation

- Interpretation as described earlier involves a lot of overhead, as bytecodes are fetched, decoded and then dispatched to their corresponding actions.
- Example: Step through a bytecode sequence (a+b) in your interpreter and see how much effort is done compared to the useful work.

1. Threaded code

- One way to decrease the cost of interpretation is to fetch and decode only once, compute an address for an action routine, and then use the addresses directly in subsequent executions.

- This is called *threaded code*, and was originally described in [Bell 73] — as a technique for a compiler!

This piece of code is really difficult to be implemented by C. since C is still a high-level language.

Before and after threading
Example: x = y + z

bytecode program (in data memory):

```
push local0
push local1
add
pop local2
```

high-level code of interpreter:

```
loop {
  get next bytecode b
  switch(b) {
  case push: …;
  case add: …;
  case pop: …;
  …
  }
}
```

threaded program (in data memory):

```
.word
&pushLocal0
&pushLocal1
&add
&popLocal2
```

interpreter loop (RISC-like assembly):

```
Loop: load [Rpc], Rtarget
      jump Rtarget
```

actions (assembly):

```
pushLocal0:
   ; push local0 on to stack
  add Rpc,BPW,Rpc
     ;inc Rpc by BytesPerWord
  jump Loop
```

Eliminating the loop

threaded program in data memory:

```
&pushLocal0
&pushLocal1
&add
&popLocal2
```

interpreter loop (assembly):

not needed; just set `Rpc` to first program address and jump through it

actions (assembly):

```
pushLocal0:
  ; push local0 on to stack
  add Rpc,BPW,Rpc
  load [Rpc], Rtarget
  jump Rtarget
```

Eliminating the loop

threaded program in data memory:

```
&pushLocal0
&pushLocal1
&add
&popLocal2
```

interpreter loop (assembly):

not needed; just set `Rpc` to first program address and jump through it

actions (assembly):

```
pushLocal0:
   ; push local0 on to stack
   add Rpc,BPW,Rpc
   load [Rpc], Rtarget
   jump Rtarget
```

This three-instruction sequence is often a single CISC instruction

Control flow

Fig. 2 Flow of control: interpretive code.

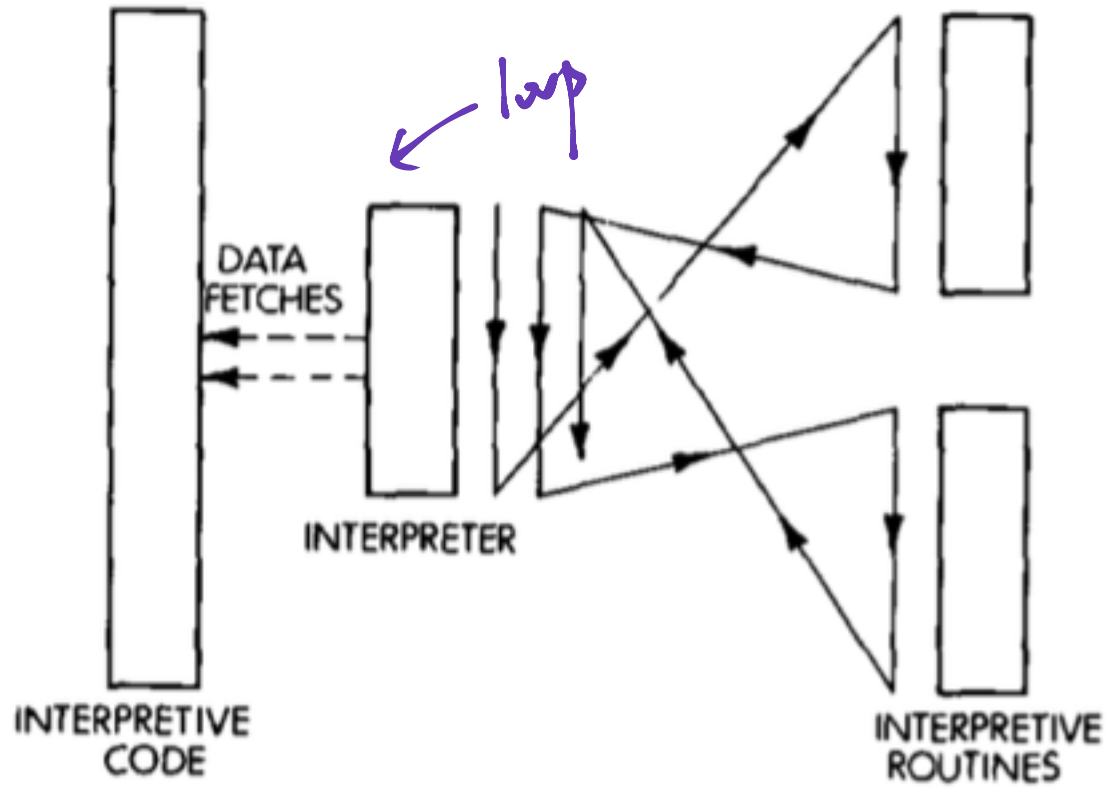
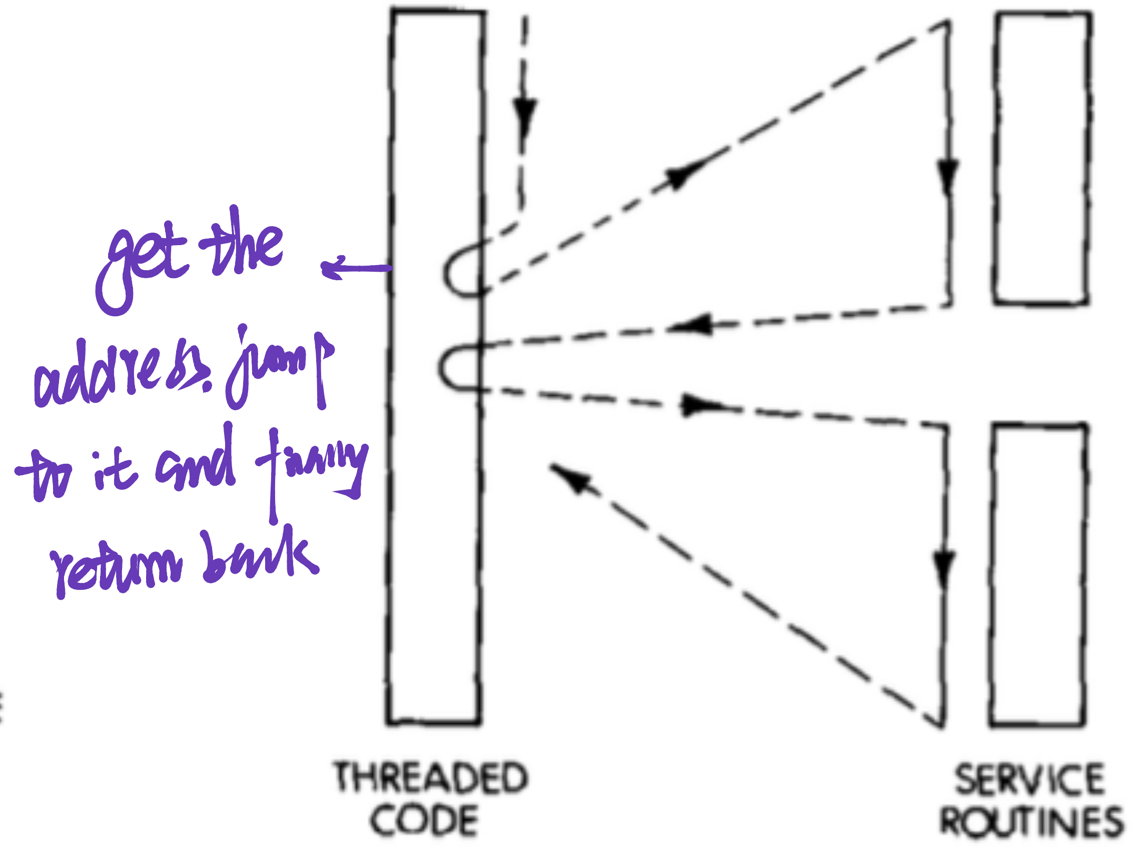


Fig. 3. Flow of control: threaded code.



Control flow

Fig. 2 Flow of control: interpretive code.

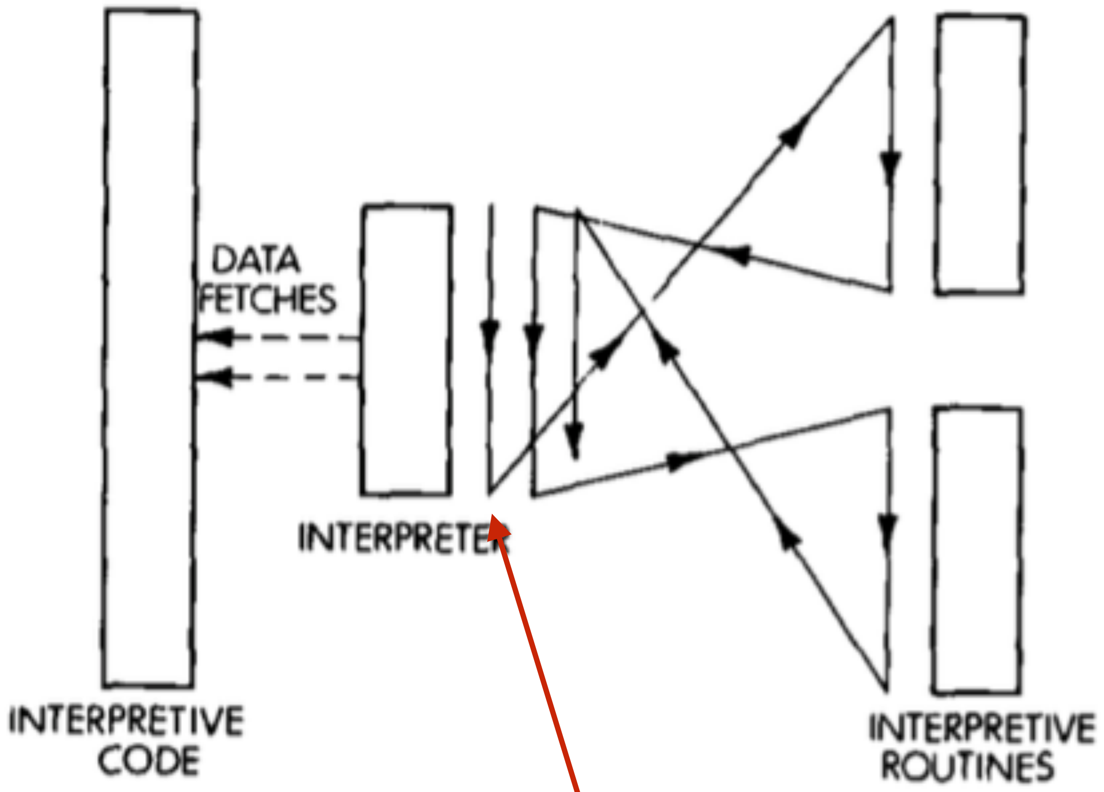
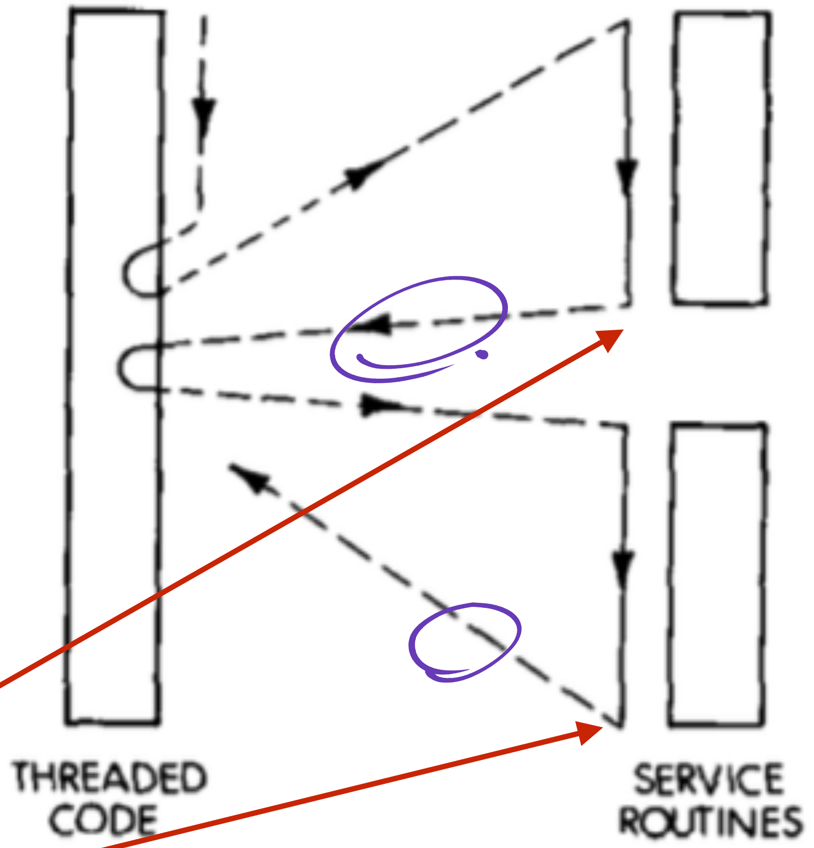


Fig. 3. Flow of control: threaded code.



very hard to predict

Alternative using executable code (calls)

program (directly executable):

```
call pushLocal0
call pushLocal1
call add
call popLocal2
```

interpreter loop: not needed

actions (assembly):

```
pushLocal0:
  ; push local0 on to stack
  ret
```

ret → predicatable

In this scheme we're using native `call-ret` to elide Rpc and the associated instructions.

Manipulation of the return address is required to implement branches, or, simply insert direct branches within the instruction stream.

Q. Is this interpretation or compilation? If compilation, when did we cross the line?

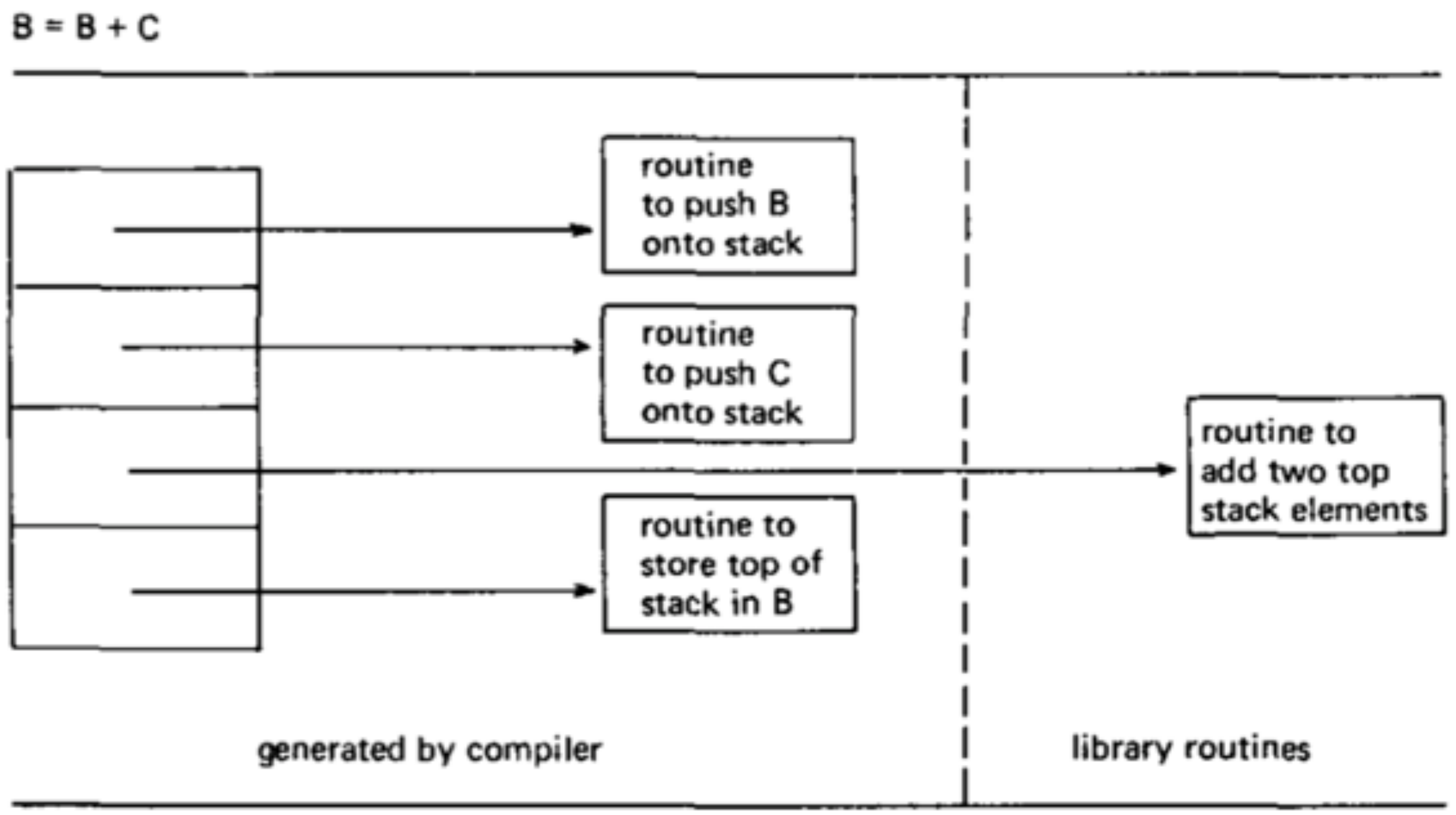
Generating the threaded code from bytecode

- To generate the threaded code, we run an abstract interpreter over the bytecode which emits the addresses of the action routines.

- We may then be able to discard the bytecode; or, ✓ we can generate threaded code into a cache and discard it as needed.

In the original paper, the technique was to be used by a compiler: it would emit the threaded code, and also generate specialized action routines (so that only the ones that were needed were emitted).

Fig. 1. Example of Direct Threaded Code.


B = B + C
routine to push B onto stack
routine to push C onto stack
routine to add two top stack elements
routine to store top of stack in B
generated by compiler
library routines

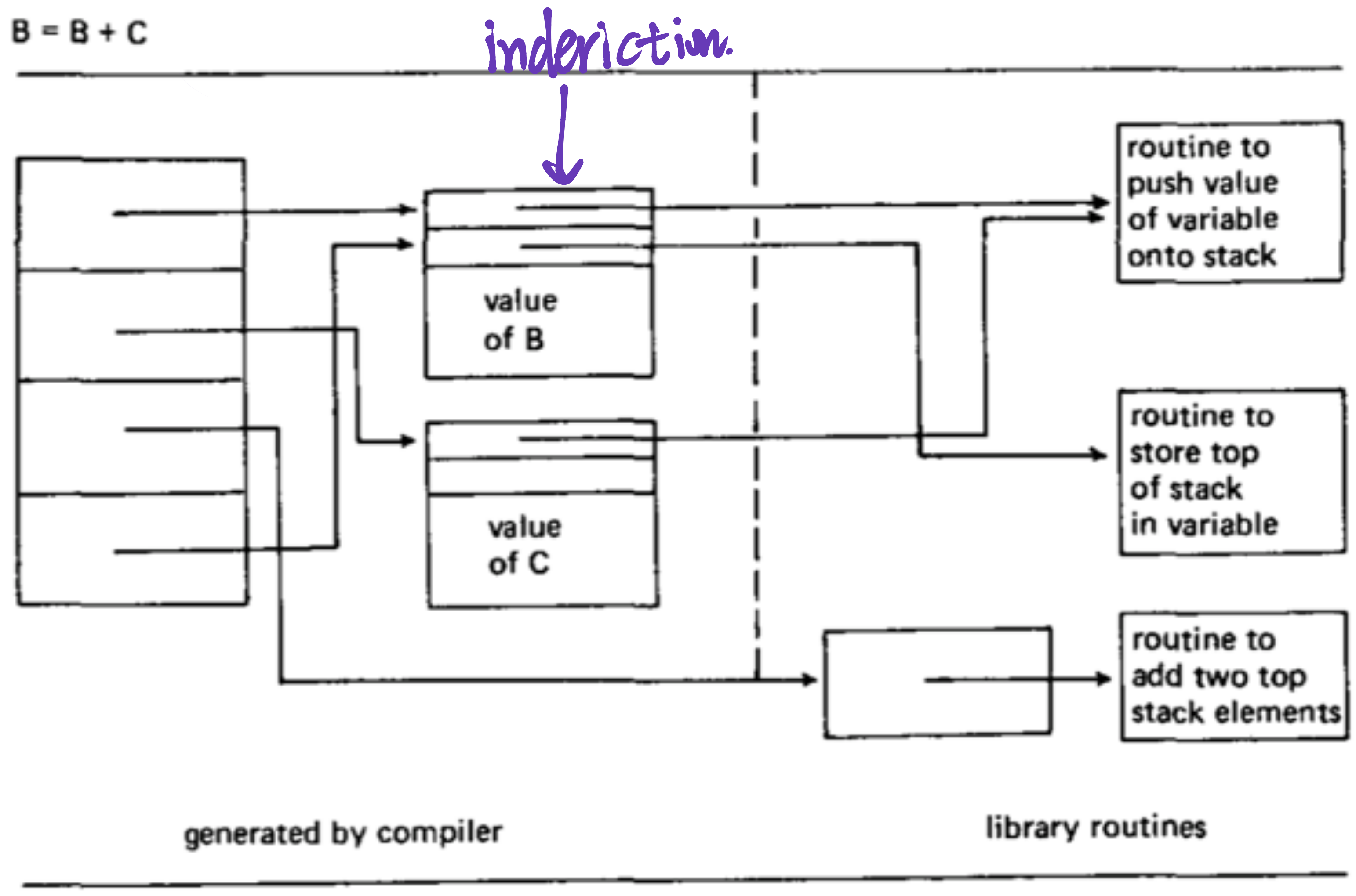
Under current case. push local 0, push local 1 will have seperate routines, but with only indexing difference.

If all the action routines have to exist before execution (e.g., we want to avoid generating code at run-time), how do you avoid a massive number of slightly different variants? (eg push local 0, push local 1, push local 2, …)

Indirect threaded code

- [Dewar 75] solves the problem by adding an indirection.

Fig. 2. Example of Indirect Threaded Code.


B = B + C
inderiction.
value of B
value of C
routine to push value of variable onto stack
routine to store top of stack in variable
routine to add two top stack elements
generated by compiler
library routines

Indirect threaded code

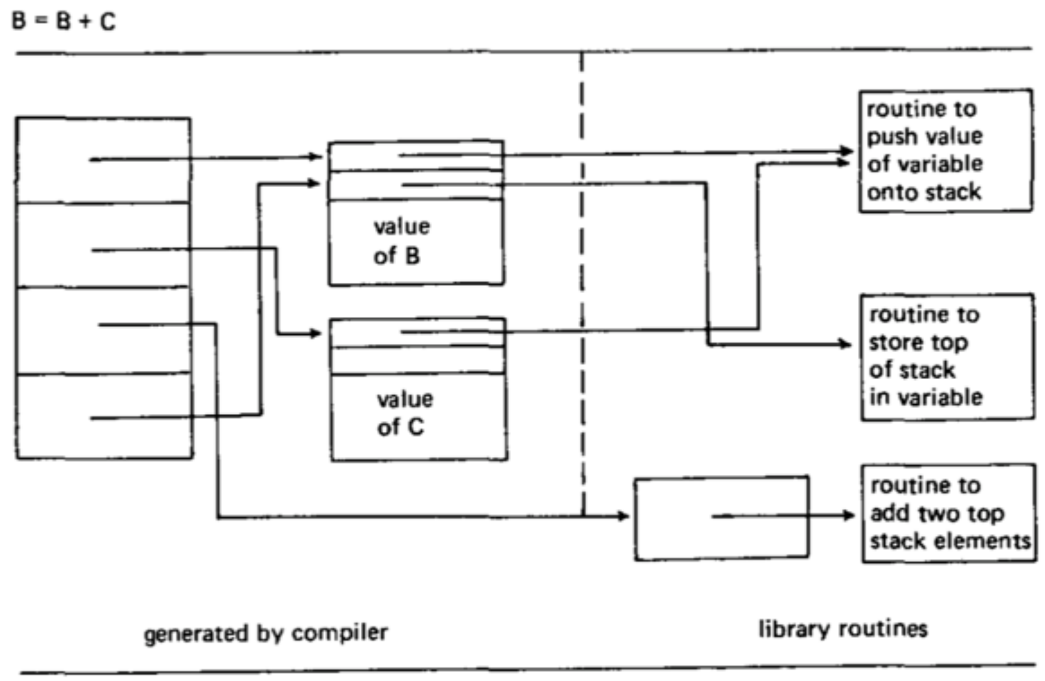
```
&pushB
&pushC
&add
&popB
…

pushB: &push
popB:  &pop
B:     .word 0

pushC: &push
popC:  &pop
C      .word 0

add: @addCore
```

Fig. 2. Example of Indirect Threaded Code.



B = B + C
value of B
value of C
routine to push value of variable onto stack
routine to store top of stack in variable
routine to add two top stack elements
generated by compiler
library routines

```
push:
  ;push [[Rpc]+2BPW] onto stack
  add Rpc,BPW,Rpc
  load [Rpc], Rtarget
  load [Rtarget],Rtarget
  jump Rtarget
```

Problems

- This scheme is fine for old FORTRAN (statically allocated variables) but unattractive for languages that need a function stack, or which contain complex data types — we'd need to create pointers to getters and setters for every stack slot and data object.
- Can use the "map trick" — at the cost of another indirection.

Alternatively…

- We decompose a complex bytecode into simpler pieces. E.g., instead of a branch-to-offset-n bytecode, we can push *n* onto the expression stack and then have a generic branch that takes the offset from the stack.

- We can use this trick with indirect threading, to implement, e.g., activation record getters and setters.

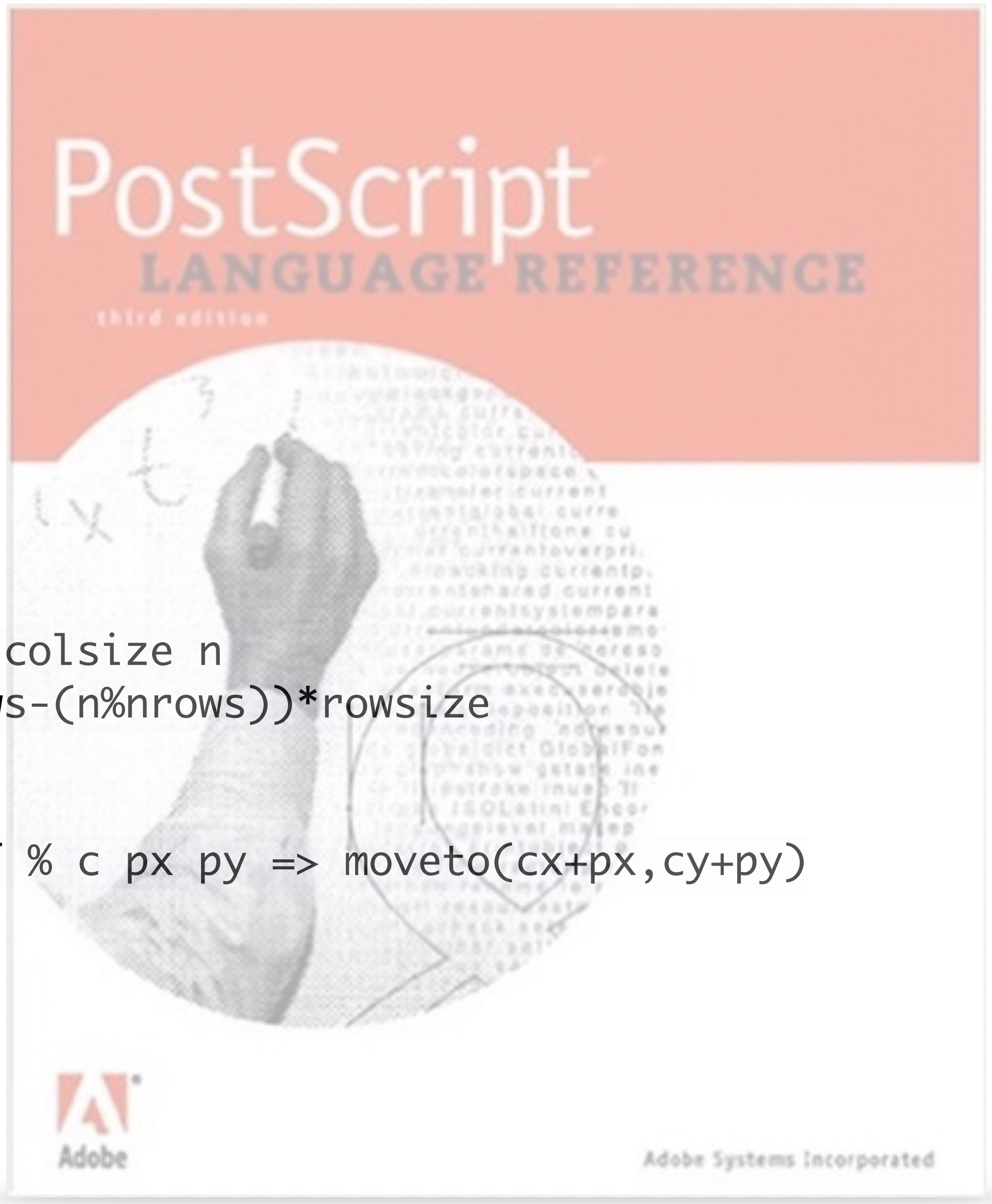
```
push-immediate-3
push-local-variable ; push local var whose
                    ; index is TOS
```

- The price to be paid is slower execution and bulkier code

Threaded code in practice

- Threaded code was adopted by the Forth language, which was at one time very popular in control applications.
- It was also adopted for PostScript, which is the underlying graphics language of PDF — so you use a threaded code interpreter every time you render a PDF.

PostScript sample



```
/ncols 2 def
/nrows 10 def

/colsize 72 4 5 32 div add mul def % 4 5/32"
/rowsize 72 1 mul def % 1"

/origin{dup % n => n n
 nrows idiv colsize mul exch % n n => (n/nrows)*colsize n
 nrows mod nrows exch sub rowsize mul % n=>(nrows-(n%nrows))*rowsize
 moveto 0 -27 rmoveto}def

/nextpos {cy add exch cx add exch moveto pop}def % c px py => moveto(cx+px,cy+py)
```

2. Stack caching

- In a bytecoded stack machine, most bytecodes manipulate the item on the top of the stack.
- Implemented naively, this results in each bytecode doing a load and/or store to the current top of stack.
- ✓ One optimization is to cache the top of stack in a register, and rewrite the bytecodes to use the register; this usually means writing in assembly.

Variants on stack caching

- If the underlying hardware partitions its register set among different types (e.g., integer, floating point, address) then it may be impractical to choose a single register for the top-of-stack cache.

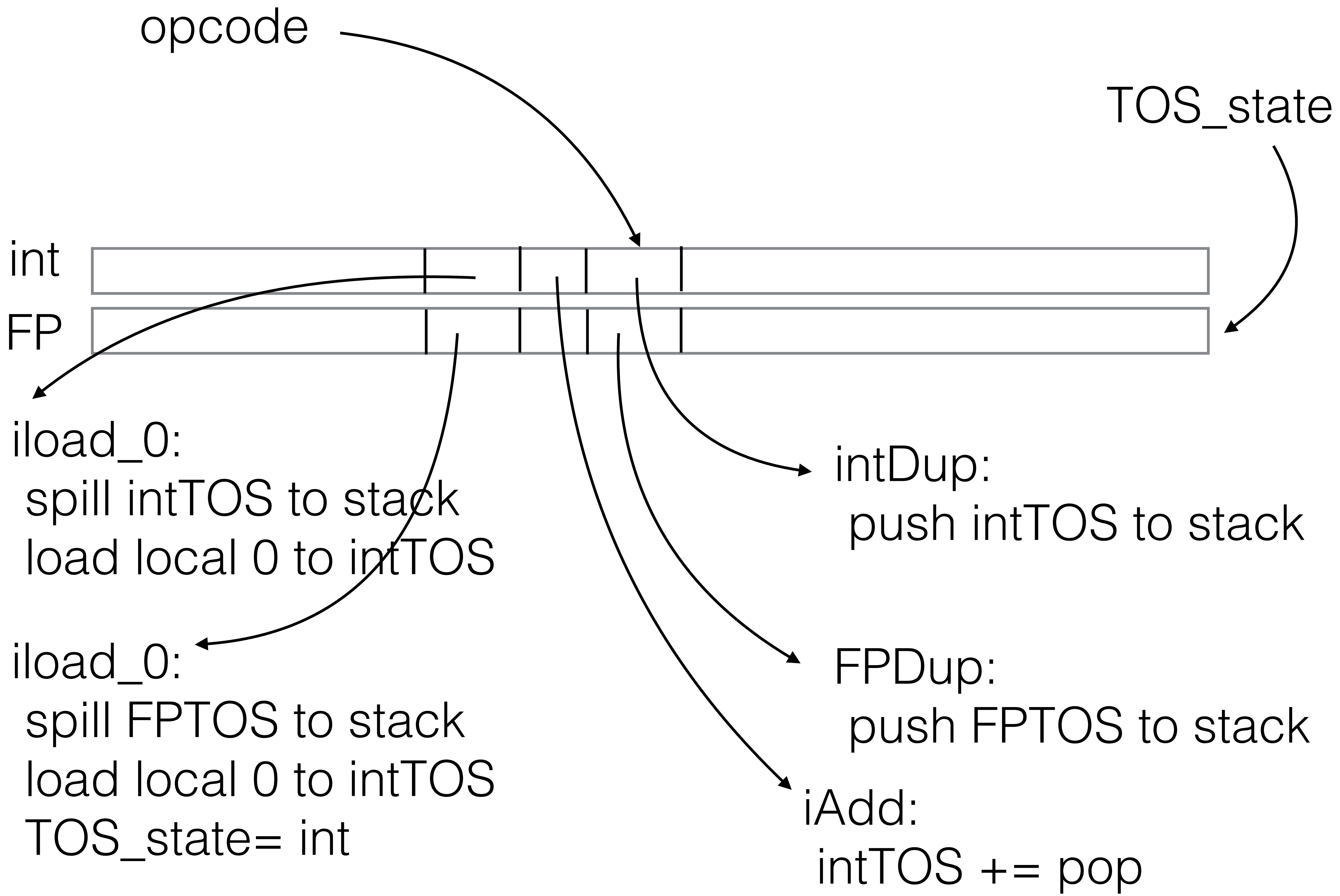
- In this case, the top-of-stack can be held in the most convenient register location based on its type. Some state-tracking code may be required; or if the bytecode set allows, this may be inferred from the abstract stack state (e.g., in Java, the type of the top of stack is always constant for a particular bytecode instruction).

Multiple dispatch tables

- Even if the TOS state is known for each bytecode, how do you take advantage of that?

- Example: consider two *dup* bytecodes in a Java class file; one is known to dup an int, the other a float, but they have the same opcode.

- One technique is to have multiple complete bytecode dispatch tables, one per state, indexed by opcode, and to switch between them when necessary.



```
iload_0:
 spill intTOS to stack
 load local 0 to intTOS

iload_0:
 spill FPTOS to stack
 load local 0 to intTOS
 TOS_state= int

intDup:
 push intTOS to stack

FPDup:
 push FPTOS to stack

iAdd:
 intTOS += pop
```

3. Making a bytecode instruction do more

- Another way to reduce interpreter overhead is to make each instruction do more, thereby diminishing the relative overhead of bytecode dispatch.

- In some systems, a single bytecode instruction can accomplish a great deal of work. Examples:

  - In Smalltalk-80, the BitBLT instruction is used to copy and blend rectangular areas on the display or in off-screen bitmaps

  - In R, and especially its predecessor, S, the bytecodes were seen as a glue to connect statistical operations on vectors, expressed in FORTRAN, C or C++.

Superinstructions

- Another way to achieve this is to synthesize *superinstructions* from common sequences of bytecodes.

- This can be done programmatically, by analyzing a corpus of bytecode programs to determine common patterns (pairs, triplets, and so on).

- Having done this, the bytecode compiler and interpreter can be modified to emit and execute, resp., these superinstructions; or a single pass over original bytecode can be used to transform the patterns into the new instructions.

Example

- The sequence *pushLocal x; inc; popLocal x* can be replaced by a single instruction *incLocal x*.
- This is akin to making an instruction set CISCy.
    - Bigger opcodes

4. Selective inlining

- Threaded code and superinstructions can be combined dynamically using *selective inlining* [PiumartaRiccardi98]:
- Concatenate the implementations of several bytecodes at runtime, and install a threaded code pointer to them, replacing the original bytecode entries.

compiled code:

```
void *code[] = { ...,
  &&opcode_push3,
  &&opcode_push4,
  &&opcode_add, ... };
```

opcode implementations:

```
/* dispatch next instruction */
#define NEXT() goto ****++instructionPointer

  void **instructionPointer = code - 1;
  /* start execution: dispatch first opcode */
    NEXT();
  /* opcode implementations... */
  opcode_push3:
    *++stackPointer = 3;
    NEXT();
  opcode_push4:
    *++stackPointer = 4;
    NEXT();
  opcode_add:
    --stackPointer;
    *stackPointer += stackPointer[1];
    NEXT();
  /* ... */
```

Figure 2: Direct threaded code.

```
dynamic_opcode_push3_push4_add:
  *++stackPointer = 3;
  *++stackPointer = 4;
  stackPointer--;
  *stackPointer += stackPointer[1];
  goto ****++instructionPointer;
```

Figure 4: Equivalent macro opcode for **push3**, **push4**, **add**.

Details of selective inlining

- Assumes code to be copied is relocatable;
- Cannot contain relative branches outside the sequence;
- Control flow changes can only happen in the last bytecode; and
- Bytecodes that are branch targets must be at the head of a sequence.
- Example: consider only basic blocks.
- Can cache and reuse the generated macro opcodes

Interpreter Generation

- Writing an efficient interpreter is pretty much impossible in a high-level language; you have to control the low-level details, many of which are not expressible, even in C.
- Another approach is to write an interpreter generator in a high-level language, which emits the machine code of the interpreter.
- Or, invent a DSL (Domain Specific Language).
- See, e.g., vmgen [EGKP02]

Putting it all together: the HotSpot interpreter(s)

- The HotSpot^{TM} JVM contains an interpreter generator that uses many of these techniques:
  - It uses stack caching with different states for each JVM type and per-type dispatch tables
  - The machine code of the bytecode action routines is generated at startup from descriptions in C++ (thinly disguised assembly). Instruction selection is done based on the actual CPU model.
  - It used to do selective inlining

Last word on interpretation

Applying all these tricks will definitely speed up an interpreter, but it will never approach the speed possible in optimizing compilation…our next topic.