

Java™ on the IA64 Architecture

Michał Cierniak

Intel Corporation

Microprocessor Research Lab

August 22, 2000

(Java is a trademark of Sun Microsystems, Inc.)

Outline

- Design overview
- GC
- Stack unwinding
- JIT

This is a 4-hour tutorial compressed to 1 hour.

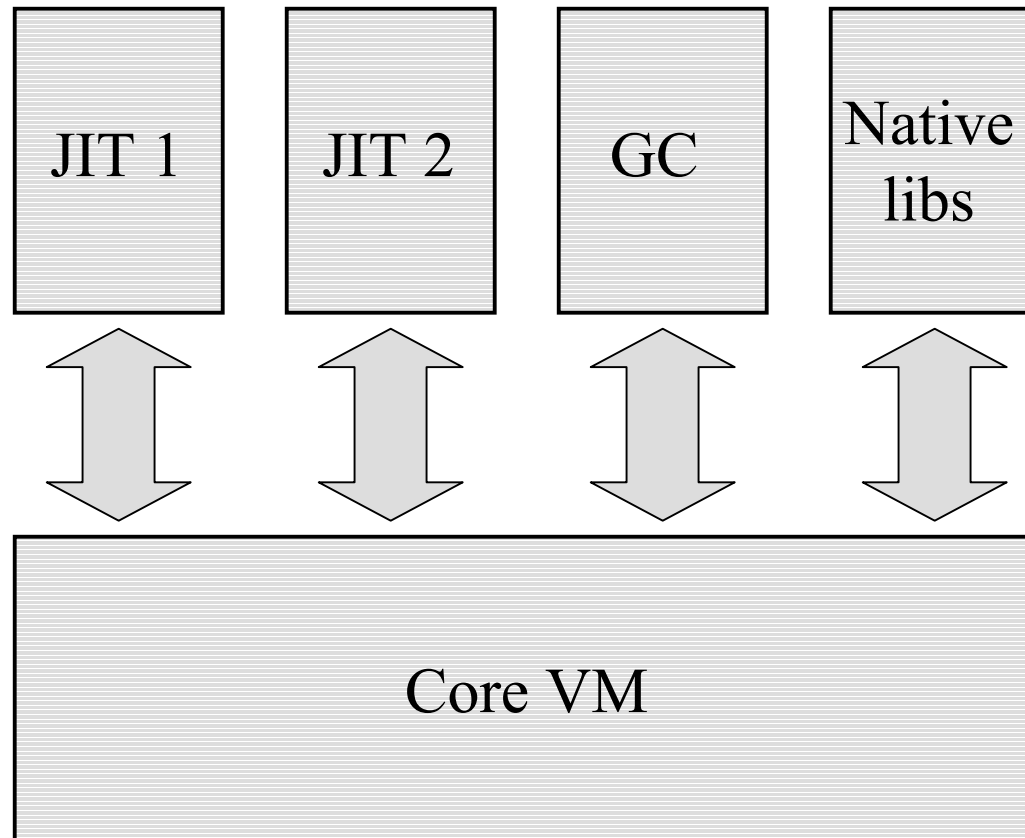
More information (IA32)

- *Practicing JUDO: Java Under Dynamic Optimizations*, Cierniak, Lueh and Stichnoth. PLDI 2000.
- *Support for Garbage Collection at Every Instruction in a Java Compiler*, Stichnoth, Lueh, and Cierniak. PLDI 1999
- *Fast, Effective Code Generation in a Just-In-Time Java Compiler*, Adl-Tabatabai, Cierniak, Lueh, Parikh and Stichnoth. PLDI 1998

More information (IA64)

- *Cycles to Recycle: Garbage Collection on the IA-64*, Hudson, Moss, Subramoney and Washburn. ISMM 2000.
- *The IA-64 Architecture Software Developer's Manual*.
- Documentation is also available for download on the web at:
<http://developer.intel.com/design/ia-64>

Virtual Machine framework



Multiple JIT Support

- Two JIT compilers:
 - Fast code generator
 - Optimizing compiler
- Used to perform dynamic recompilation
- More about it later

Allocation

- Rule 1 - Common case must be fast
- Rule 2 - Counting instructions is not enough
- Rule 3 - see Rule 1

Allocation

- Strategy
 - Minimal restrictions imposed on JIT
 - Do not require gc safety at every allocation point!
 - Fold all checks into single overflow check
 - (Fake) object size > nursery size
- Tactics
 - Overflow on fast case returns NULL
 - Caller checks and advances to GC safe point

Slow Path Allocation

- Allocation succeeds (or throws exception)
- Speed secondary
- Class loader fakes object size
- Deals with Fixed Object Space
- Deals with finalization
- Deals with weak/soft/phantom refs
- Not discussed further

General Approach

- Per CPU allocation area
 - better than per thread since number of threads is not bounded
- Assumes that every context switch will do a little work. (Talk to OS guys.)
- Fast short code sequence is goal
- Allocation pointer in register

Allocation sequence

:: swt and swf are predicate registers, always holding opposite values
;;; swt means a task switch and resumption have happened
;;; swt and swf are thread-local
:: ap is the allocation pointer
:: vt is the new object's vtable pointer value
:: sz is the new object's size in bytes
:: np receives the address of the new object

top:

(swf) mov np = ap ;; indicate address of new object
(swf) st8 [ap] = vt ;; store vtable pointer
(swf) add ap = ap, sz ;; bump allocation pointer
(swt) br retry ;; task switched, so retry whole sequence

....

retry::; reset pred regs

cmp.eq swf, swt = r0, r0 ;; set swf true, swt false

br top ;; go retry

With limit check

:: st, sf, ap, vt, sz, and np are as before

:: lp is the limit pointer

top:

(swf) mov np = ap ;; indicate address of new object

(swf) st8 [ap] = vt ;; store vtable pointer

(swf) add ap = ap, sz ;; bump allocation pointer

(swf) cmp.le swf, swt = ap, lp ;; merge limit test result into swf, swt

(swt) br retry ;; task switched or past limit

....

retry:

cmp.eq swf, swt = r0, r0 ;; set swf true, swt false

cmp.le 0, pgt = ap, lp ;; redo limit test to discriminate

(pgt) br.call rp = gc ;; call gc

br top ;; go retry

Clearing object

- Java requires objects to start life type safe
- Zero is a legal value for all types
- So `gc_malloc` returns only cleared objects
- Some OS initially clear space

When to clear an object

- After GC evacuates an object
 - GC latency is increased
- When an object is allocated
 - Cost of setting up loop dominates clearing cost
- When a nursery is made available - this wins
 - Allows clearing of large contiguous space
 - Remember memory fence after clearing
 - MTRT and Jess see 2% improvement

Write Barriers

- Generational GCs focus on small area of heap
- Uses stack maps for stacks
 - JVM tracks globals
 - Class structures in fixed object space
- Track all slots holding pointers to focus area
 - Scanning entire heap too expensive
- Write barrier remembers where pointers are

Card Marking

- 1 register for card table base, a right shift of the object base, plus an byte store marks card
- GC scans marked cards
- Summarize pointers found into rem set
 - Or re-mark card with relevant train/car id
- Clear card
- If card table is sparse, scanning cost is reduced by scanning 8 bytes at once

Card Mark Sequence

```
;; o holds a reference to the object modified
;; f is the offset of the field being updated
;; p is the reference being stored
;; ct holds the virtual base of the card table:
;;; the location that would hold the mark for the card at address 0
add rx = o, f      ;; form field address
st8 [rx] = p       ;; store the pointer
shr.u ry = o, k    ;; form card index (k is a constant)
add ry = ct, ry    ;; form address of card byte
st1 [ry] = GR0     ;; store the constant 0 in the entry
```

Sequential Store Buffer

- 1 reserved register, a store and an increment
- Does not remove duplicates like cards
- Tag lower bits of pointer for consumer
- Use guard page or just check low bits and add link
- Example as in Trishul Chilimbi
 - Monitor temporal locality of pointer reads
 - Cluster objects based on temporal locality

SSB Sequence

```
;; o holds a reference to the object modified
;; f is the offset of the field being updated
;; p is the reference being stored; m is a mask of k low-order ones
;;; (it's constant, but too big for an immediate)
;; s is the sequential store buffer pointer
add rx = o, f           ;; form field address
andcm ry = p, m         ;; round p down to start of block
st8 [rx] = p           ;; store the pointer
cmp.lt px, py = o, ry   ;; compare source and target addresses
(px) st8 [s] = rx, 8    ;; store rx to SSB, increment s by 8
```

Publication Safety

- Given a newly allocated object how does one ensure that the fields hold type safe values?
- Solution 1 - only allocate in zeroed areas
 - But then we have a null vtable
- Solution 2 - publish using a st.rel
 - ld.acqs are not needed since the st.rel will ensure initial field value is globally visible prior to object publication

Publication Safety

:: Thread T1
;; v has the vtable value
;; p has the object address
;; g points to the global
st8 [p] = v
st8.rel [g] = p

:: Thread T2
;; g points to the global
;; p gets the object address
;; v gets the vtable value
ld8.acq p = [g]
ld8 v = [p]

.acq not needed

Stack Unwinding

- Performed completely in SW
- Advantage:
 - The same code works for NT/VC++ and for Linux/gcc
- Disadvantage:
 - Cannot reuse native tools (e.g., debuggers)

Stack Unwinding: Issues

- Multiple JIT's
- Native Java methods
- Runtime support functions

Unwinding: Multiple JIT's

- Stack frame layout is only known to the JIT

```
virtual void
```

```
unwind_stack_frame(Method_Handle method,  
                    Frame_Context *context);
```

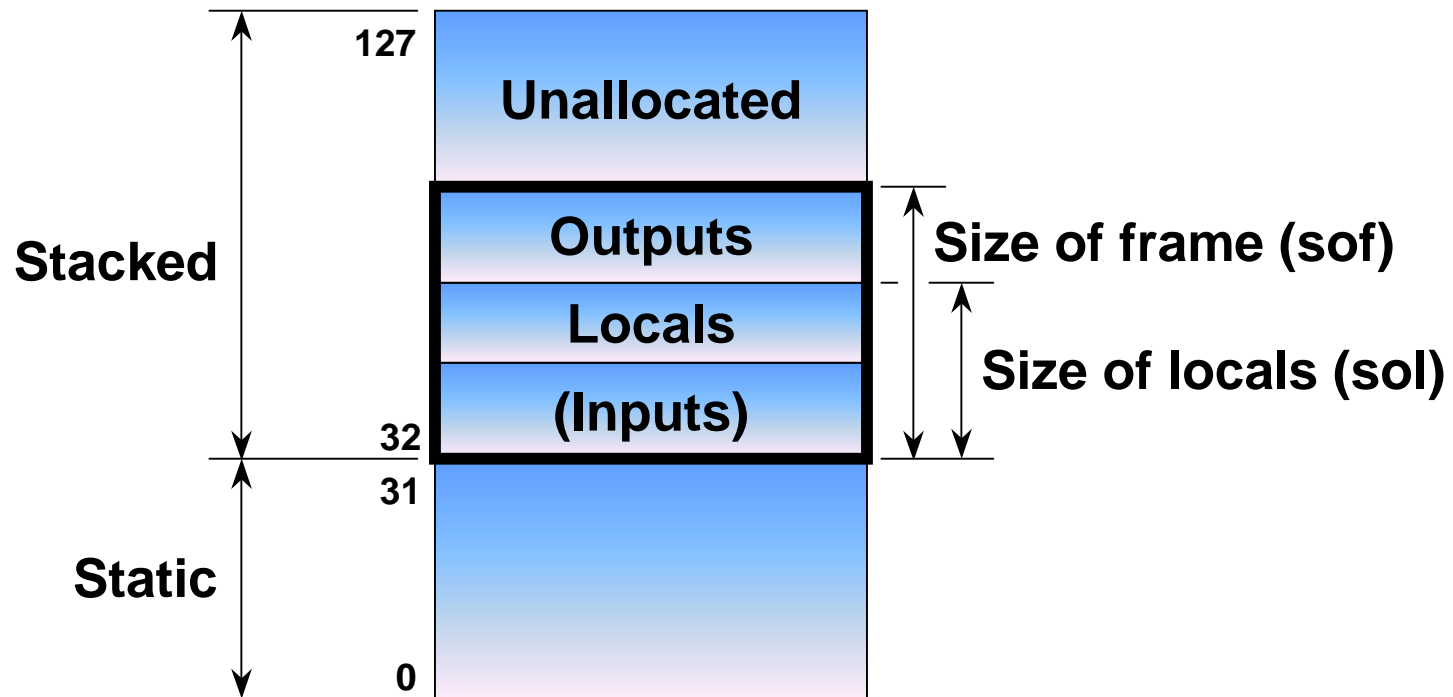

Unwinding: Multiple JIT's

```
JIT_Specific_Info * jit_info;  
jit_info = methods.find(ip);  
...  
JIT *jit = jit_info->get_jit();  
Method *method = jit_info->get_method();  
  
jit->unwind_stack_frame(method, context);
```

Stack Unwinding: Native Methods

- We assume no cooperation from the compiler used for native methods (“C”).
- `ar.bsp` is saved on every transfer from Java to C.

General Registers & Stack Frame

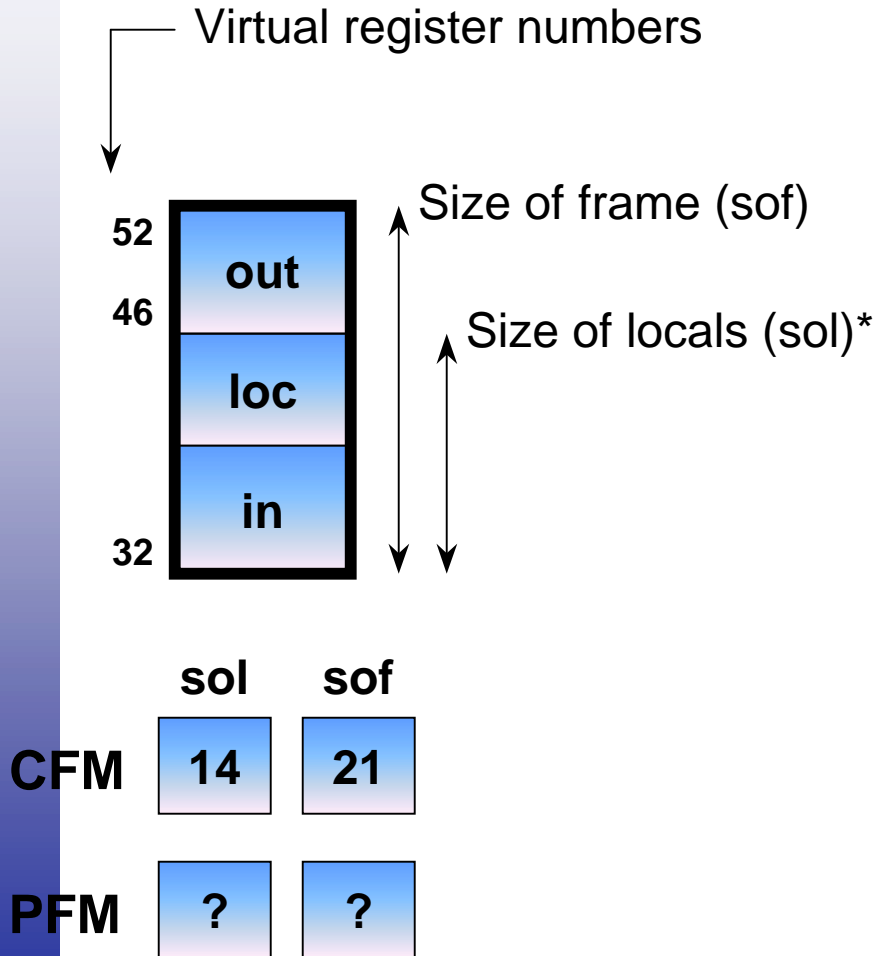


Current Frame Marker (CFM)

sol

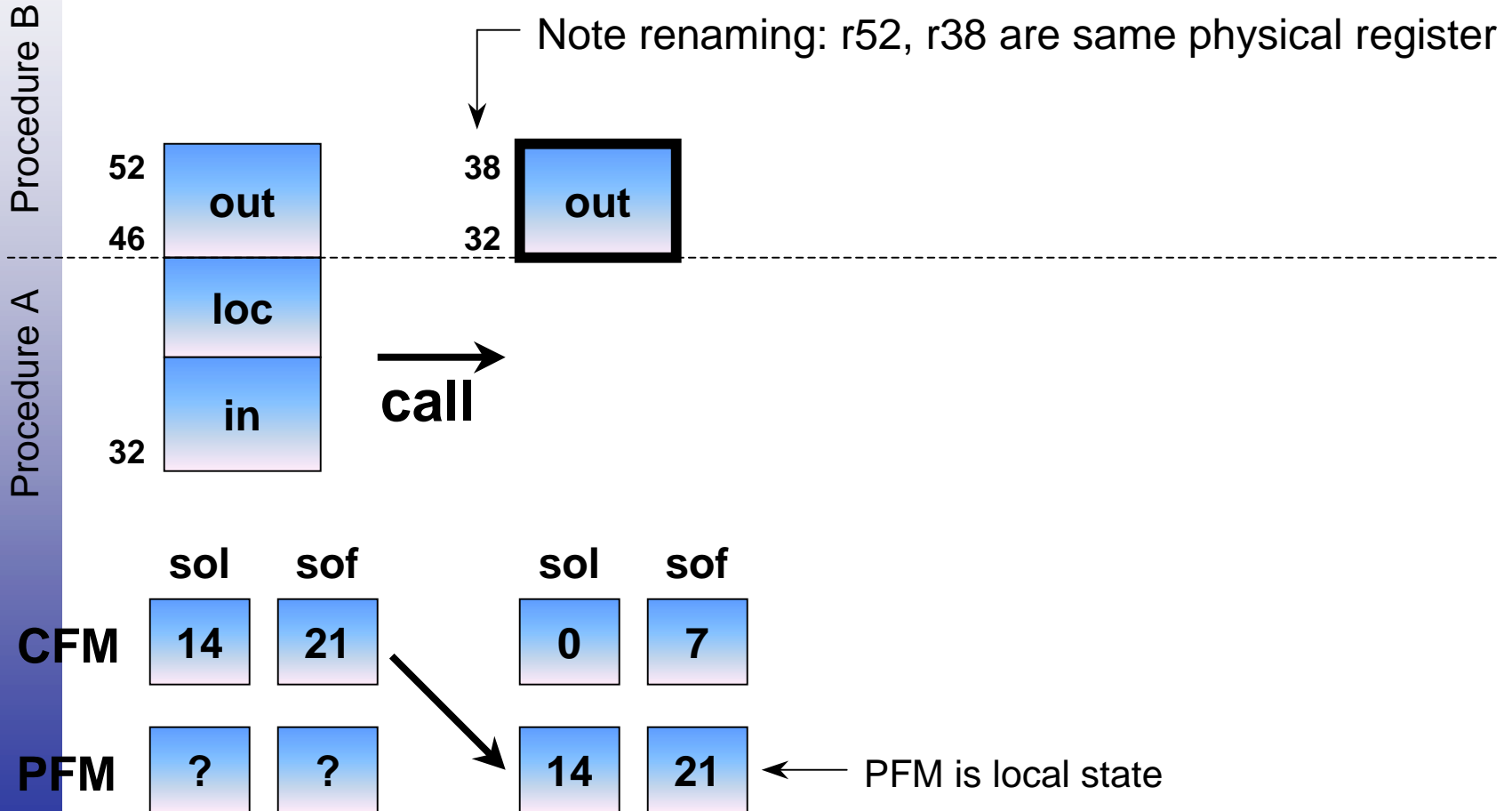
sof

GR Stack Frame: Example

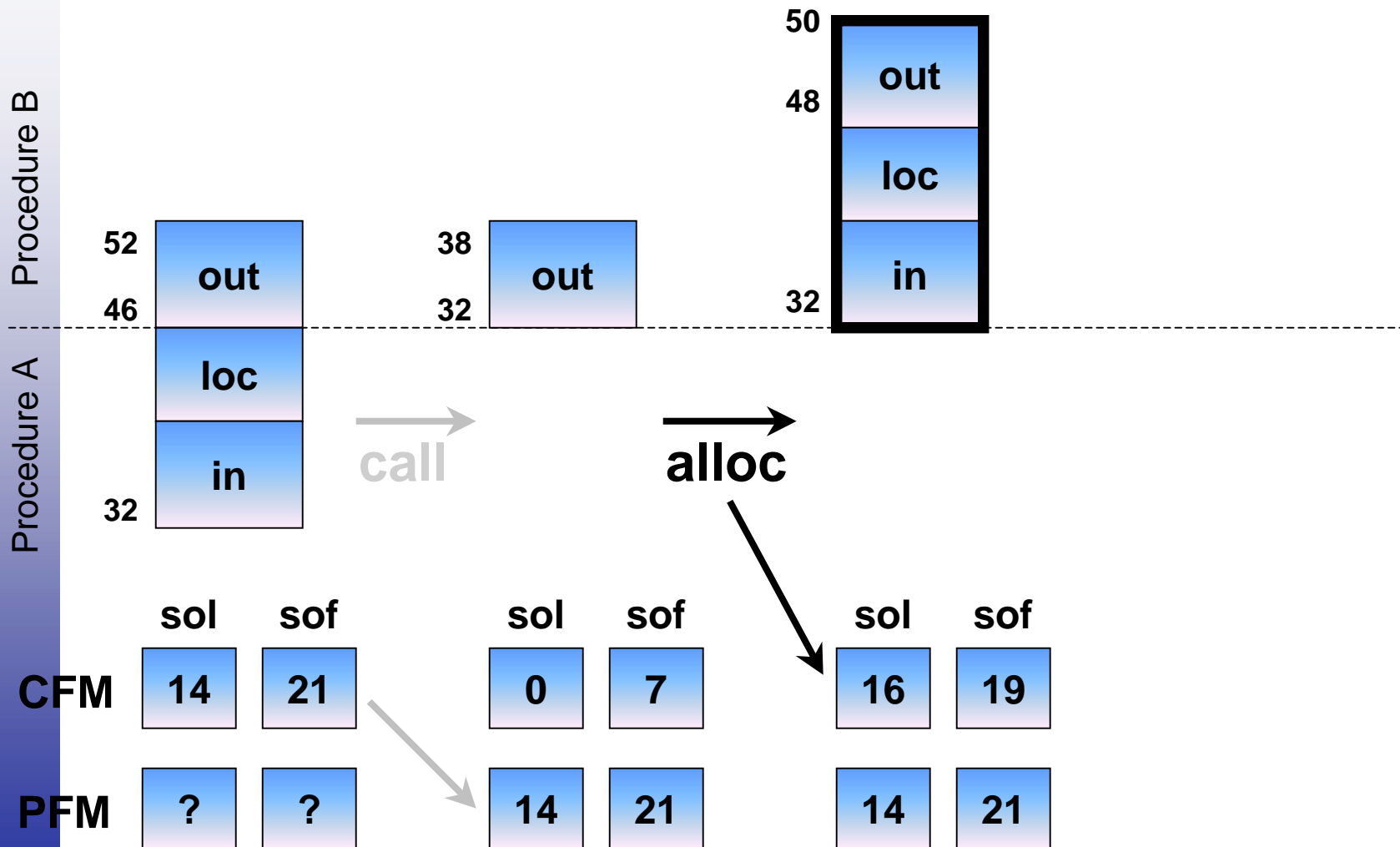


*Includes inputs

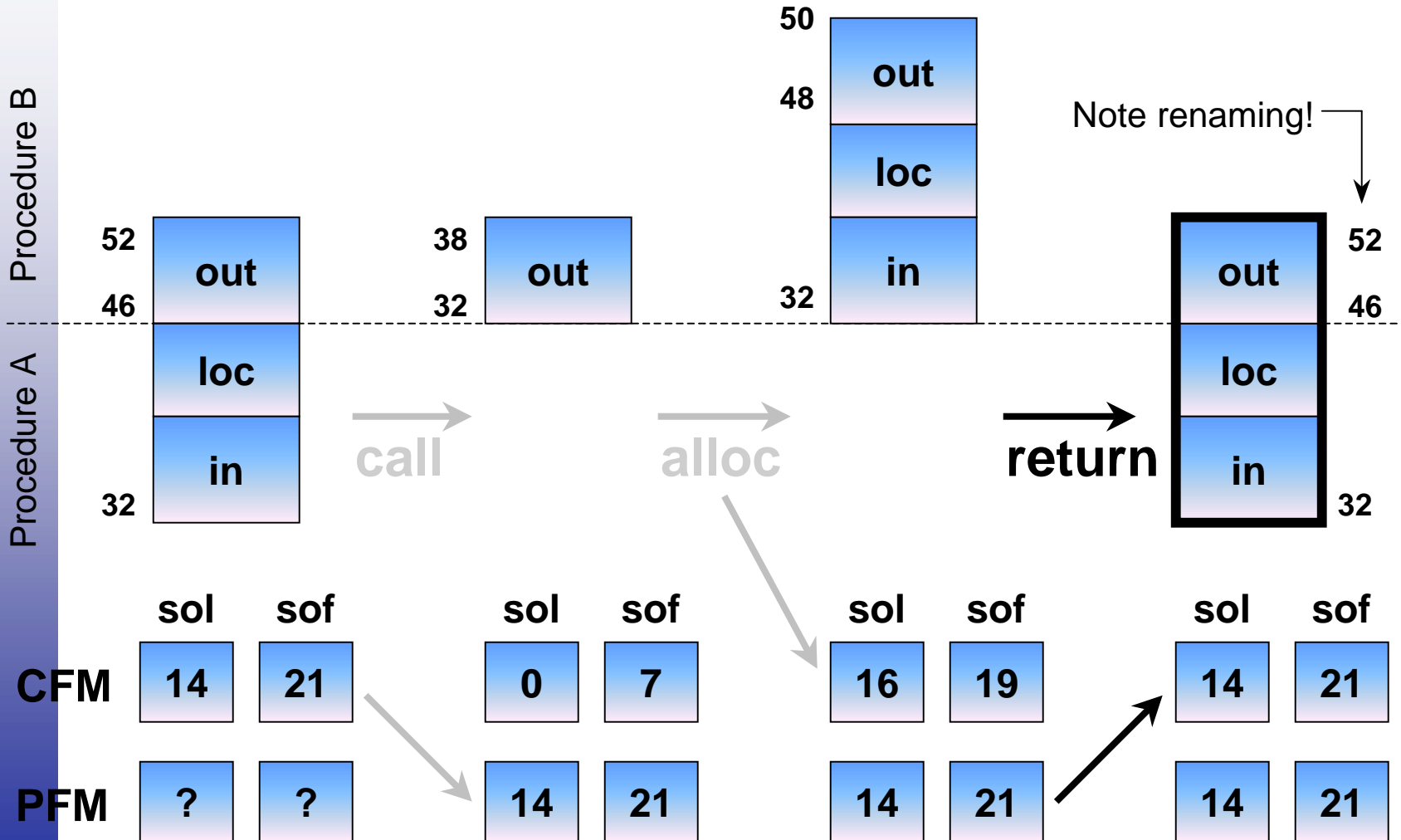
GR Stack Frame: Call



GR Stack Frame: Allocate



GR Stack Frame: Return



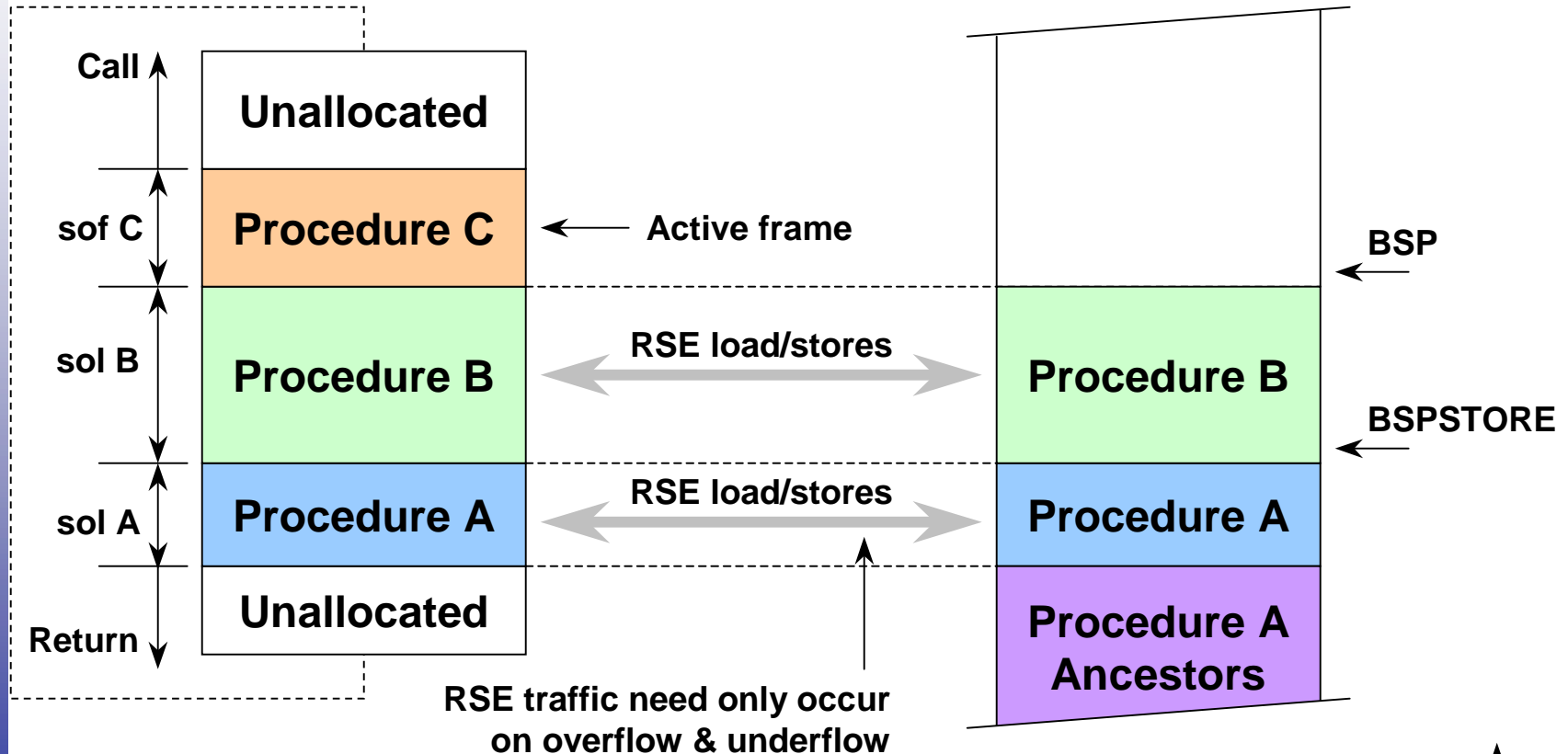
Register Stack Engine / Backing Store

- Register stack: finite physical depth but infinite virtual depth
 - Number of physical registers is ≥ 96
- “Overflow”
 - When the processor runs out of physical registers (during an alloc), the RSE automatically spills registers to memory (backing store)
- “Underflow”
 - When a procedure whose register stack was spilled to backing store returns (br.ret), the RSE restores the registers from the backing store

Register Stack Engine: Backing Store

Physical Stacked Registers

Backing Store

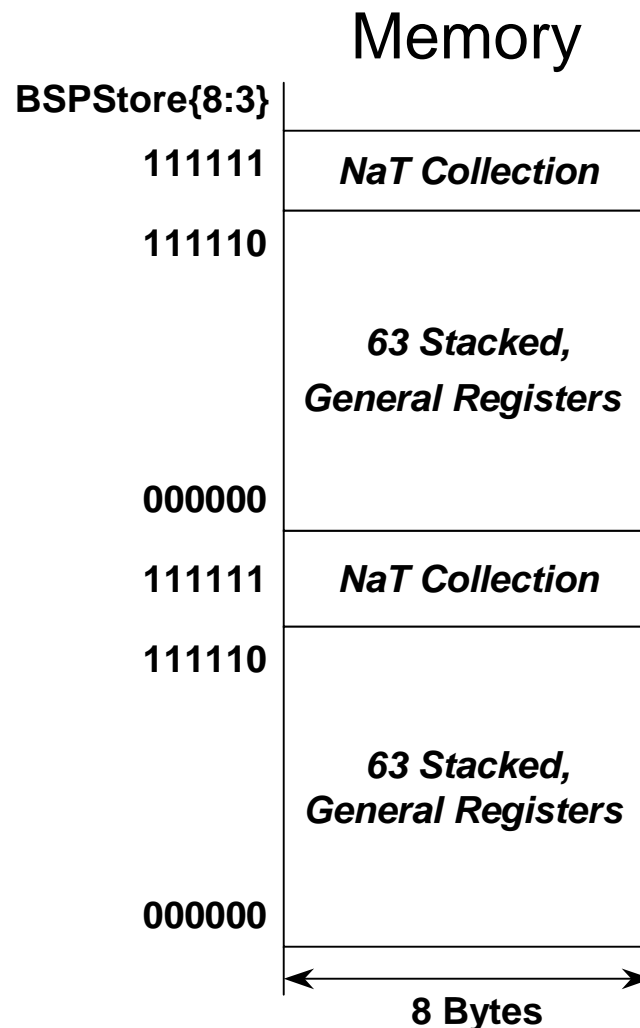


Procedure A calls Procedure B calls Procedure C

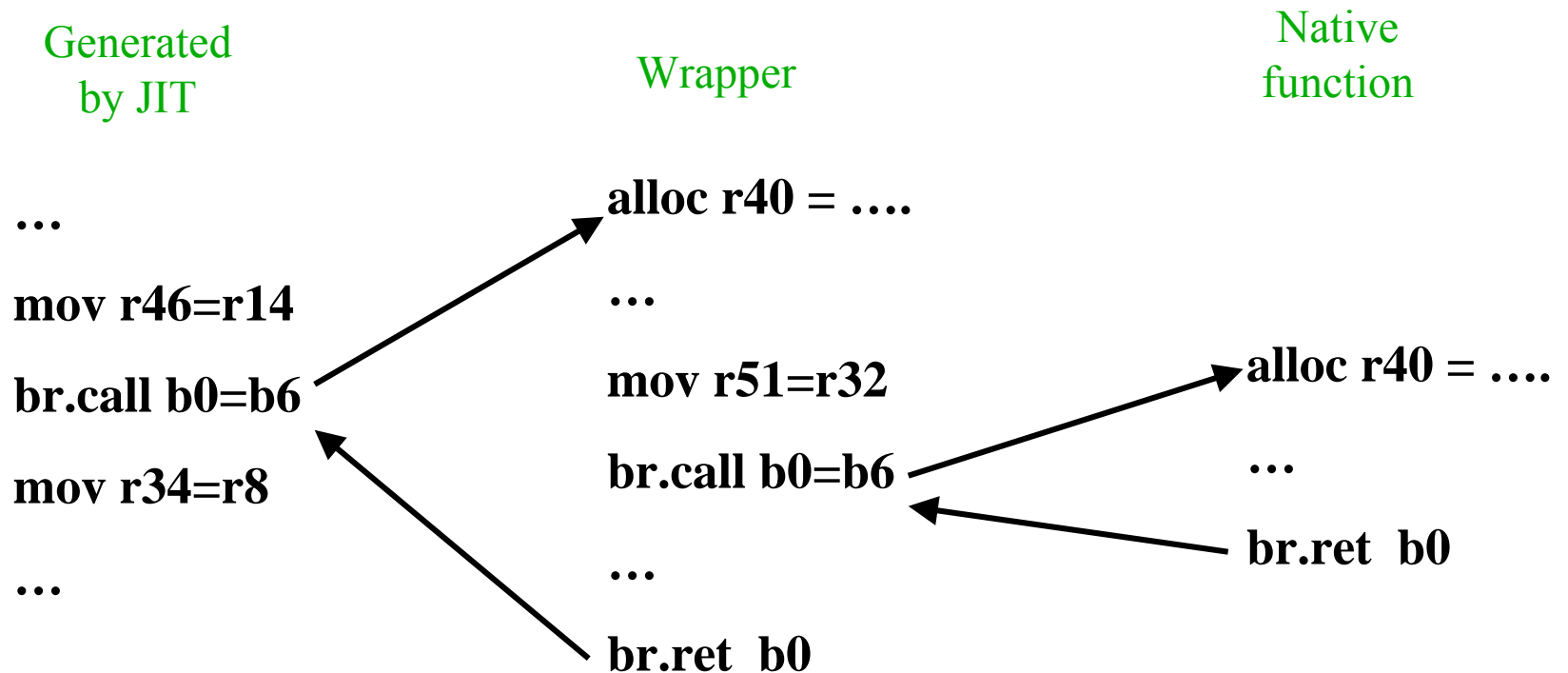
Higher register & memory addresses ↑

Register Stack Engine: NaT bits

- RSE responsible for saving NaT bits
 - Spilled/filled in groups of 63
 - Every 63 register saves/restores
 - NaTs are collected in RNaT register
- When $\text{BSPStore}\{8:3\} == 111111$
 - RNaT register is stored



Native Methods Wrappers



Native Wrapper Outline

- Create JNI handles
- Save unwind info
- Enable GC
- Call the function
- Disable GC
- Restore unwind info
- Unhandle the result (optional)

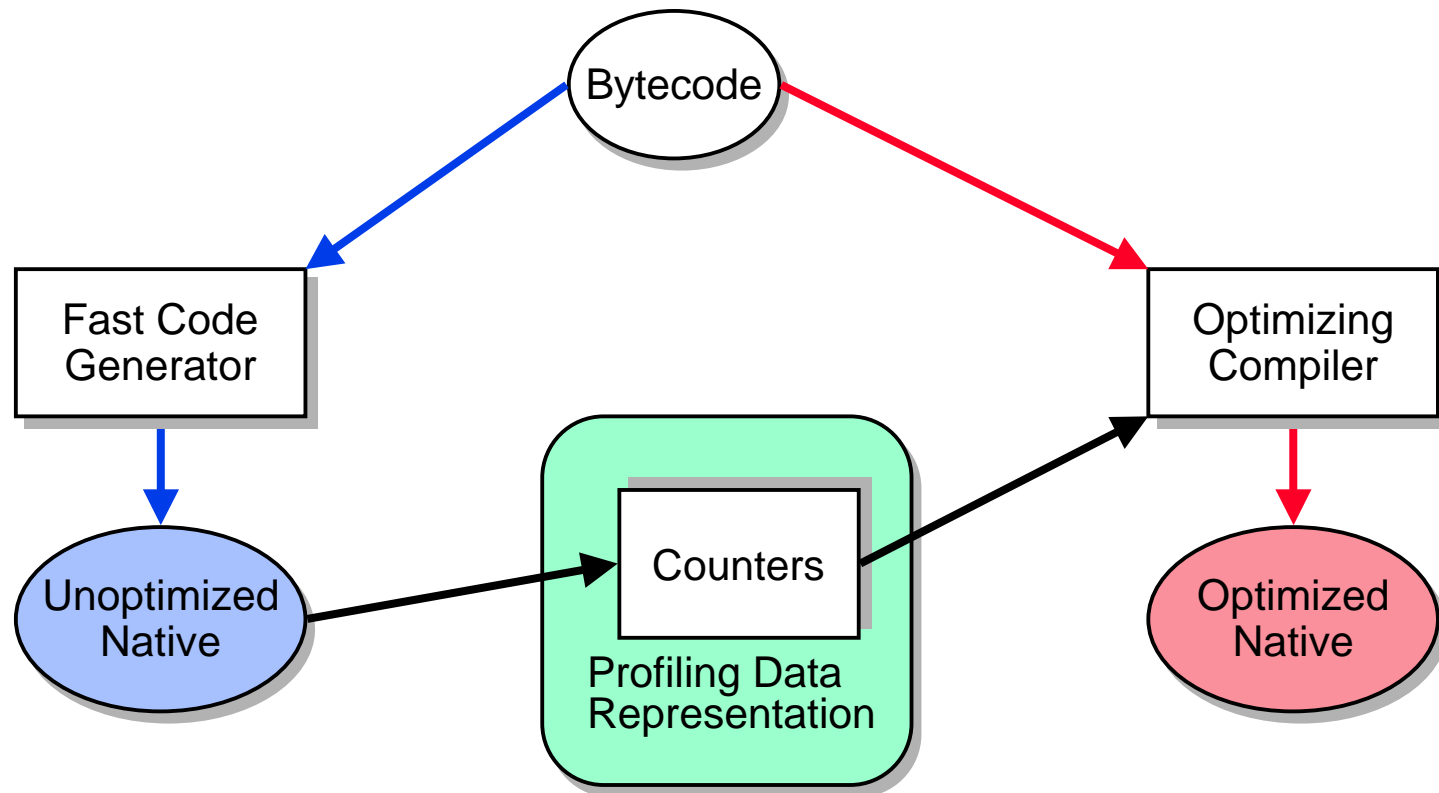
Dynamic Recompilation

- Translate bytecodes to native code at run time
 - Compilation time part of performance equation
 - Full global optimizations not always justified
- Adaptively and selectively recompile

Java recompilation strategies

- Lightweight optimization
 - Fast compilation time
 - Reasonable good performance
- Heavyweight optimization
 - Slow compilation time
 - Good performance
- 90-10 rule
 - 90% methods: lightweight
 - 10% methods: heavyweight

Structure of dynamic recompilation



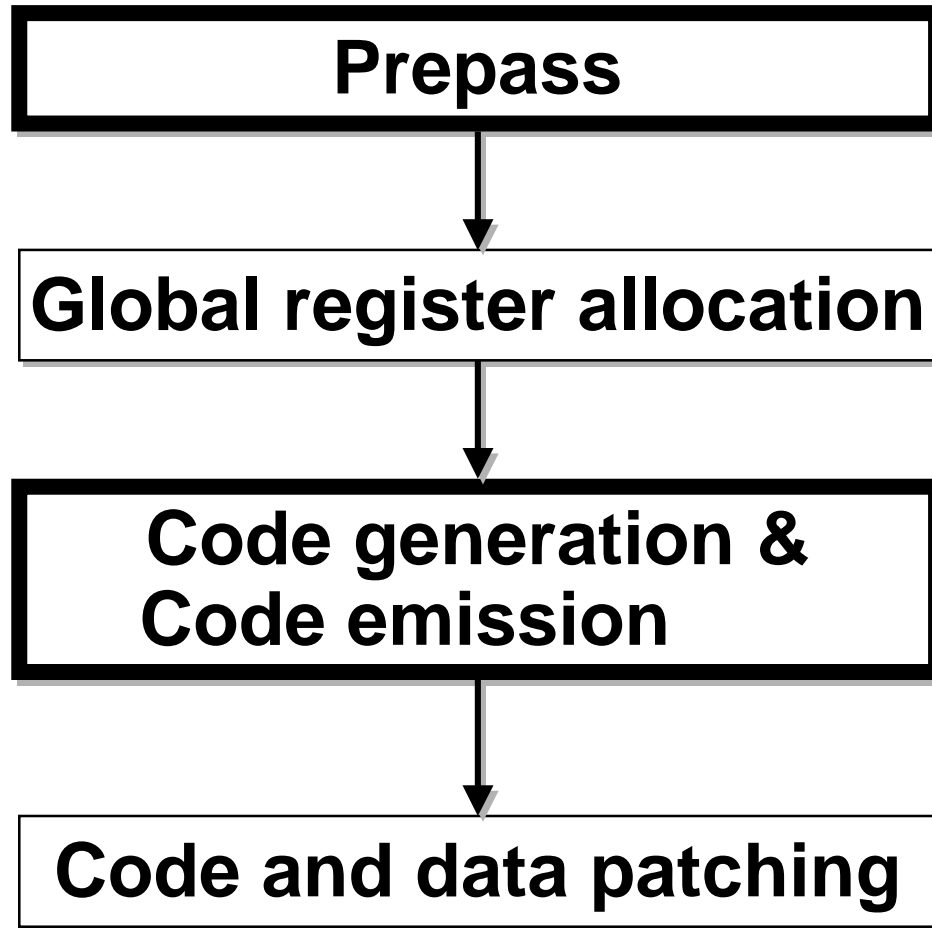
Fast code generation vs. interpreter

- When to recompile
 - Interpreter
 - reduce compilation time
 - degrade performance dramatically
 - recompilation window is narrow
 - Fast code generation
 - increase more compilation time
 - improve performance dramatically
 - recompilation window is wider
- Debugging
 - Debugging optimized code is hard
 - Running on interpreter mode may not be acceptable

Fast code generator

- Fast code generation
 - 2 linear-time passes over bytecodes
- No explicit IR
 - No control flow graph or inst. list
- Fast global register allocation
 - No interference graph
 - Direct mapping
- GC support

Structure of fast code generator



**Inserting
profiling code**

Prepass

- Basic block boundaries
- Static reference counts of local variables
- Max number of output parameters
- Stack depths at the end of blocks
- Is leaf method
- Estimated iA64 code size of each block
 - Predication

Allocate registers for local variables

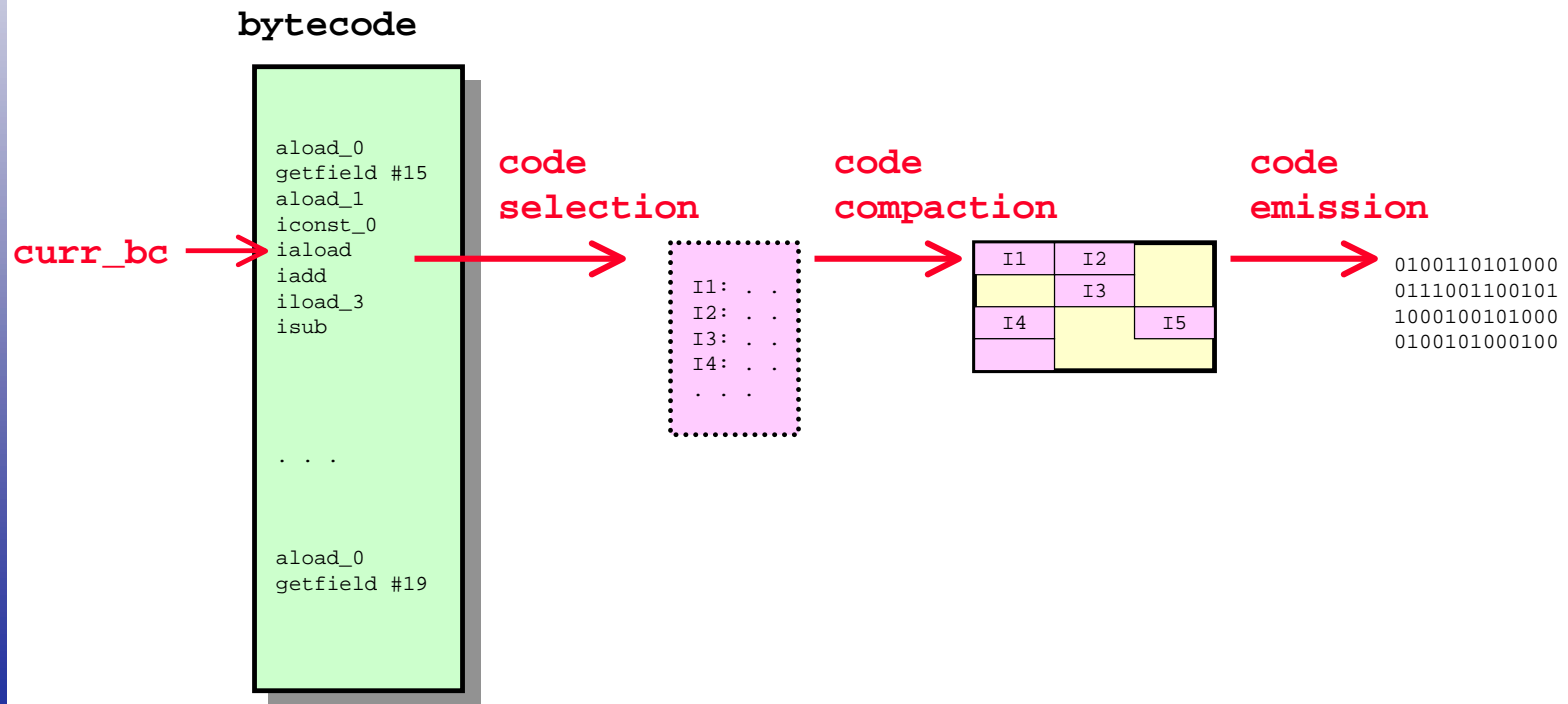
- Assign registers to variables with highest static reference counts
- May not want to assign registers to all variables
 - Reduce number of `local`
- Non-leaf methods
 - Assign unused `in` and `local` (stack) registers
- Leaf-methods
 - Assign scratch (caller-save) and `local` registers

Allocate registers to stack operands

- Register pool
 - Scratch, local and preserved
 - Explicitly manage registers
 - round-robin fashion
 - `get_reg()` and `free_reg(r)`
- Short life span (not across calls)
 - Use scratch registers
- Operands left on stack at the end of BB
 - Move to home locations
- Operands across call sites
 - Re-shuffle scratch registers at call sites

Code generation

- Selection, compaction and emission
 - All three are done in one pass

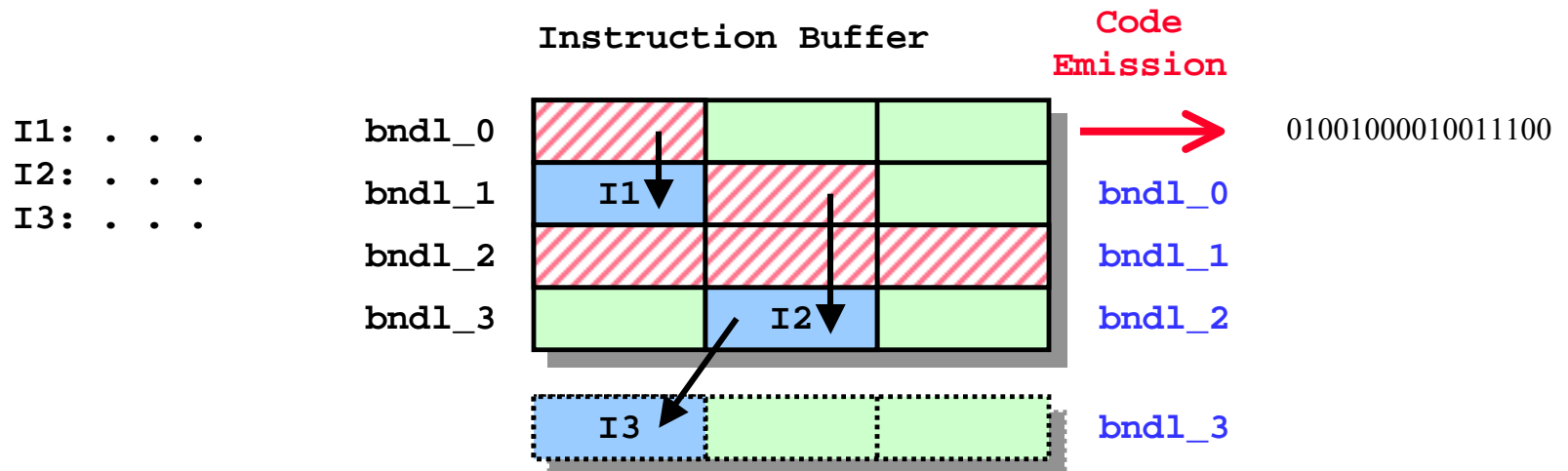


Code selection

- For each bytecode, select a sequence of iA64 insts
- Light-weight optimizations
 - Lazy code selection
 - e.g., fold immediate operands
 - Common subexpression elimination
 - Array bounds checking elimination
 - Load-after-store elimination
 - Strength reduction

Code compaction

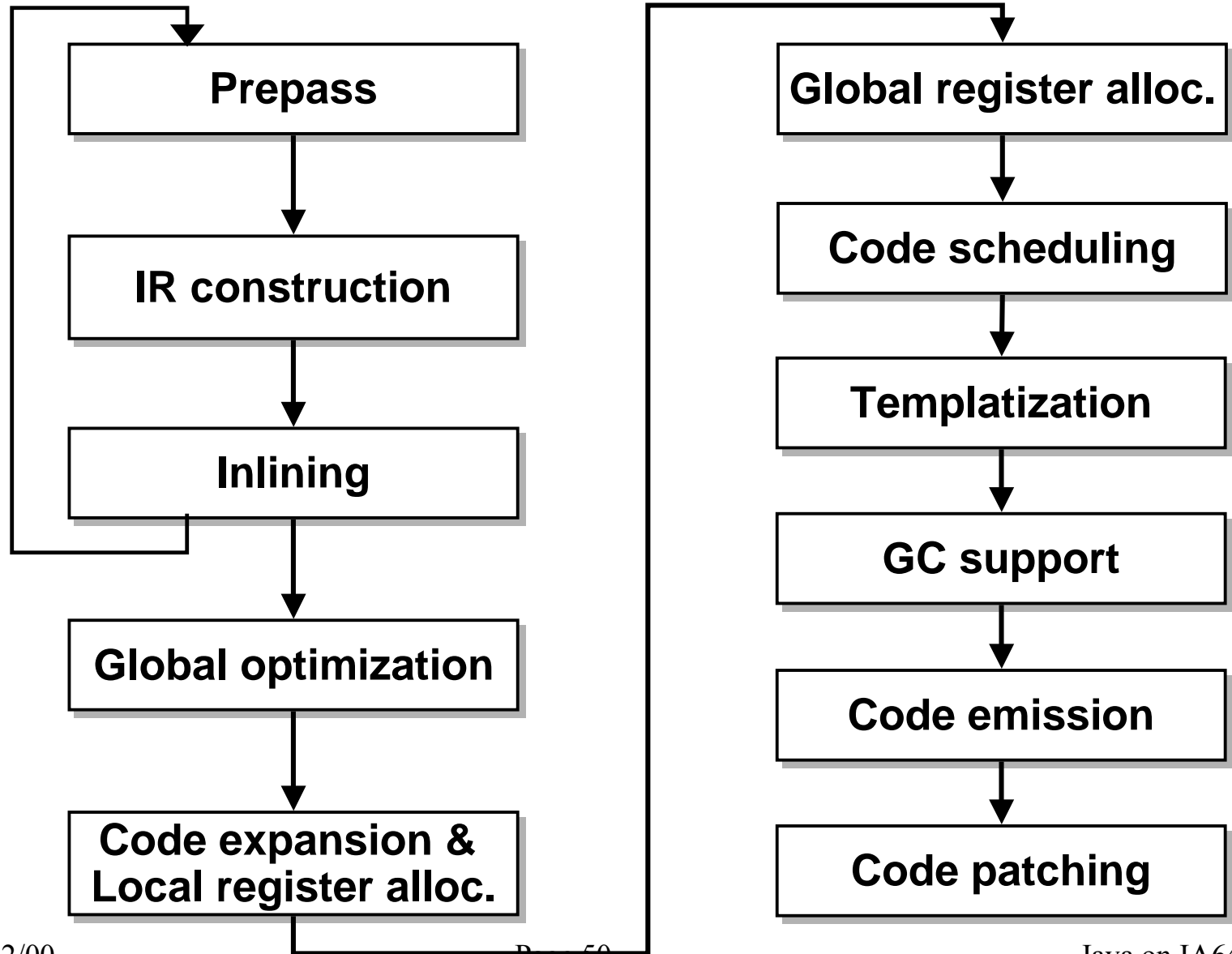
- Instruction buffer
 - Number of bundles (parameter)
 - Instructions that are not yet emitted
- Determine templates lazily
- Exchange instructions to fit into templates



GC support

- Goals
 - GC safe at every native instruction
 - Simple
- Live references
 - Local variables
 - run-time bit vector
 - schedule set/reset with astore/istore in the same bundle
 - Stack operands
 - recompile
 - track live references by code compactor
- Cache root set to avoid unnecessary recompilation

Structure of optimizing compiler



Global optimizations

- Bounds checking elimination
- Propagate exception information
 - Prove exceptions are not thrown
 - Improve code scheduling quality
- Lazy exception
 - Eliminate unnecessary creation of exception objects
 - Speed up exception throwing
- If-conversion
- Escape analysis

Code scheduling

- Dependence graph
 - Types: field, array, static, vtable, length, spill, reg, ...
- List scheduling
 - Cycle-by-cycle
 - Two I, two M, two F, or three B
 - Max 6 instructions per cycle
 - Latency
- Templatization
 - Bundle instructions and decide template types
 - Combine instructions from different cycles using mid-bundle stop bits

GC support

- Goal
 - GC safe at every native instruction
- High-level
 - Take snapshot at beginning of BB
 - Encode **delta** of each instruction
 - Coalesce contiguous instructions with no delta
- Low-level
 - Use Huffman encoding
 - Based on offline analysis of statistics
 - Dedicate a few registers for object references
- Predication
 - Check predicate reg's value during GC