Open Runtime Platform

Michal Cierniak, Rick Hudson,
Guei-Yuan Lueh and Weldon Washburn

Intel Corporation
Microprocessor Research Lab
June 2, 2001

intel®

Tutorial Outline

- ORP overview
- Synchronization and OS issues
- Garbage Collection
- JIT compilation

What is ORP

- "*A platform for bytecode system research*"
- What does that mean???
- A virtual machine with JIT and GC modules
- Currently supports Java*. We are considering support for CLI.
- Linux and Windows* (NT, 2000, XP)
- We have released an IA32 version. We plan to release ORP IA64 later this year.
- Can be downloaded from http://intel.com/research/mrl/orp

*All other brands and names are the property of their respective owners.

intel®

Why is ORP interesting

- Flexible architecture
  - Easy to plug in a new JIT or GC module
- Open source (BSD-like license)
- Good performance
  - See mailing list messages (http://groups.yahoo.com/group/orp)

Research papers

- *Using Annotations to Reduce Dynamic Optimization Time*. Krintz and Calder. PLDI 2001.
- *Sapphire: Copying GC Without Stopping the World*, Hudson and Moss. Java Grande 2001.
- *Cycles to Recycle: Garbage Collection on the IA-64*, Hudson, Moss, Subramoney and Washburn. ISMM 2000.
- *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

Information Sources

- Published papers
- This tutorial
- Some (very limited) documentation is included in ORP distribution
- Mailing list (http://groups.yahoo.com/group/orp)
- The source code itself (http://intel.com/research/mrl/orp)

How to use ORP

- You need GNU Classpath (http://www.classpath.org). A precompiled version has been made available by one of our users (http://groups.yahoo.com/group/orp/message/15)

- `orp -classpath /classpath:. Hello`
- `orp -classpath c:\classpath;. Hello`

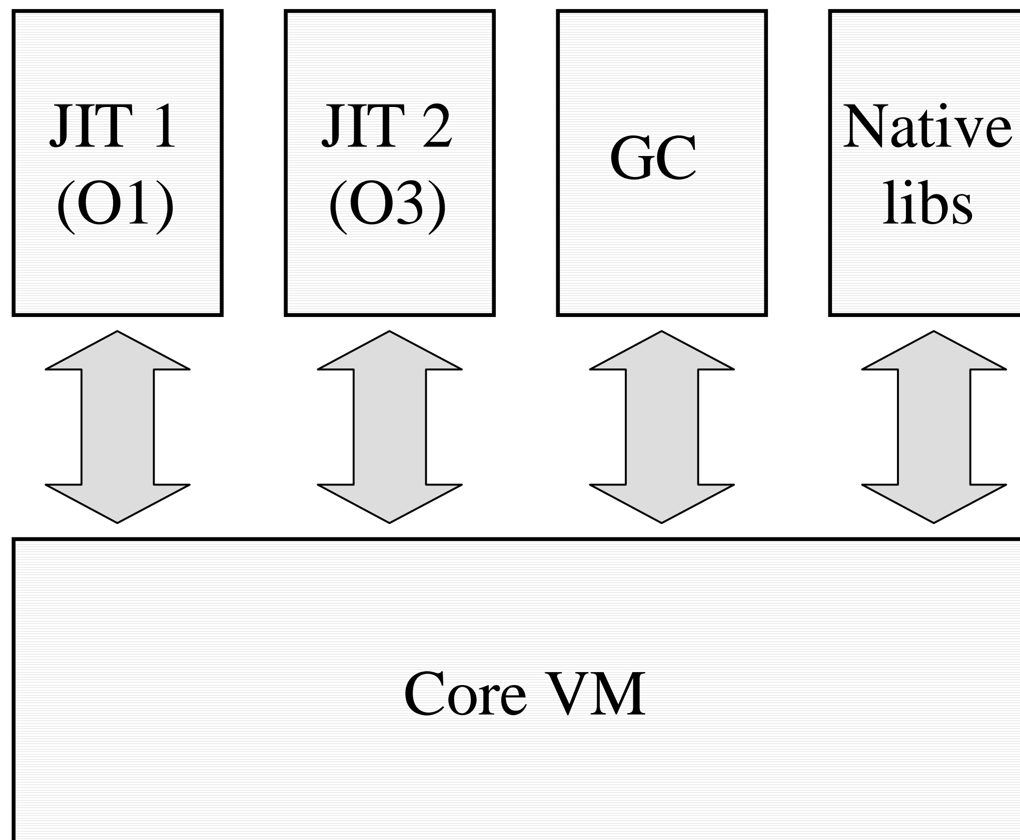
Command line options

- `-swapjit 0 1`
  - Use the O1 JIT
- `-swapjit 0 1 -jitO1a instrument`
  - Use dynamic recompilation
- `-verbosegc`
  - Print GC information
- `-ms <init> -mx <max>`
  - Set the initial and maximum heap sizes

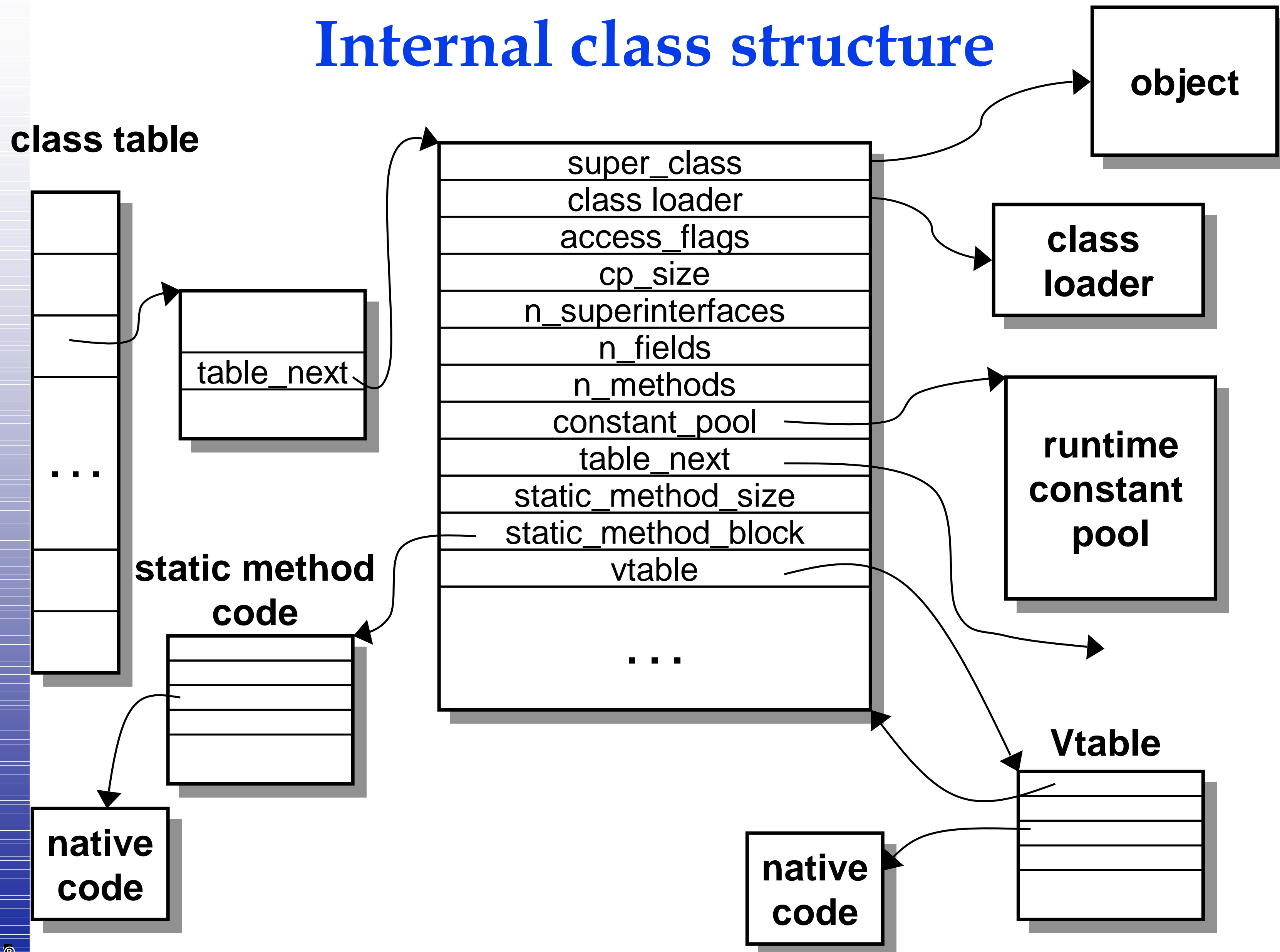
Command line options

- **`-version`**
  - Print version info
- **`-stats`**
  - Print various statistics
    (need to build ORP with –DORP_STATS)

Main Components



Internal class structure



Class

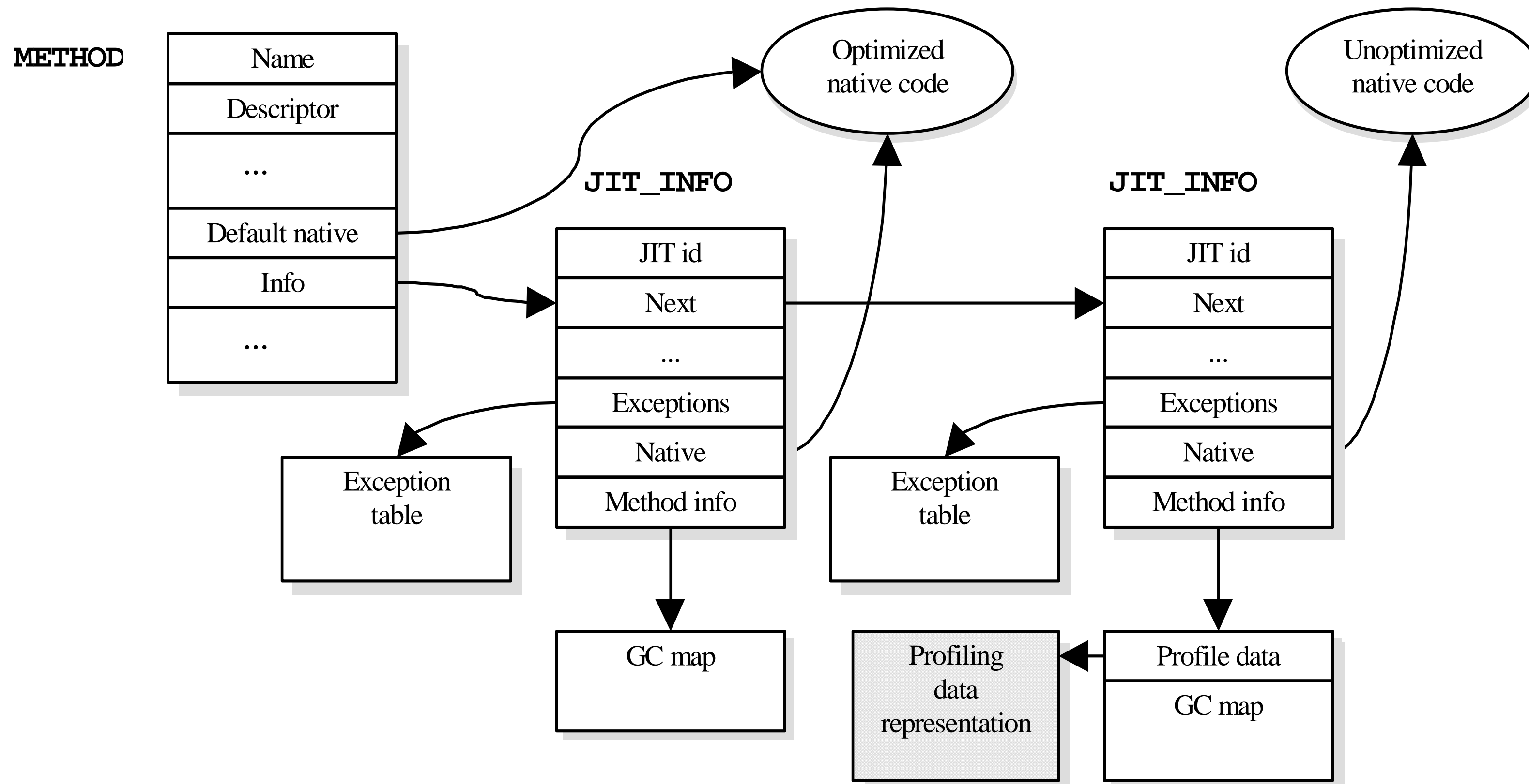
```
typedef struct Class {
    VTable  *p_vtable;
...
}
```

- Both a C data structure and a Java object of class java.lang.Class!

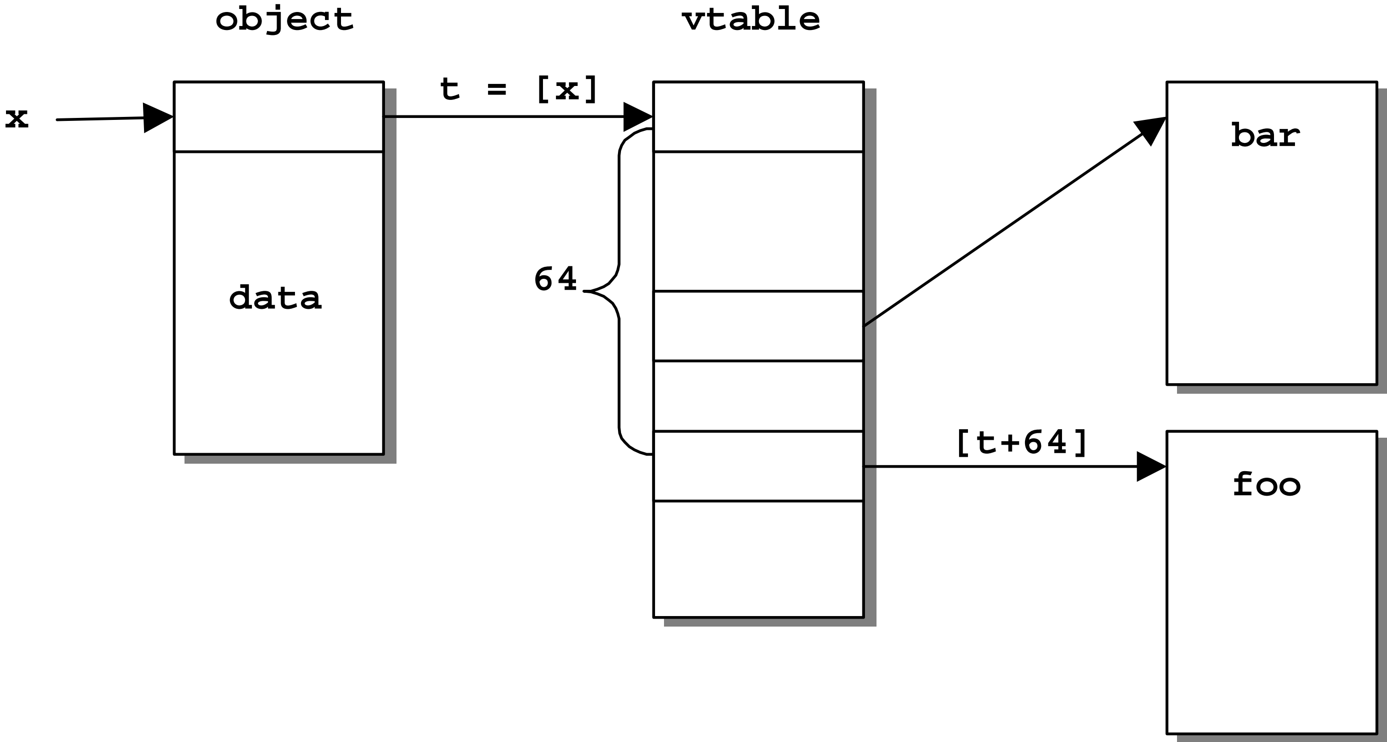
Method and Field

```
class Class_Member {
...
}
class Field : public Class_Member {
...
}
class Method : public Class_Member {
...
}
```

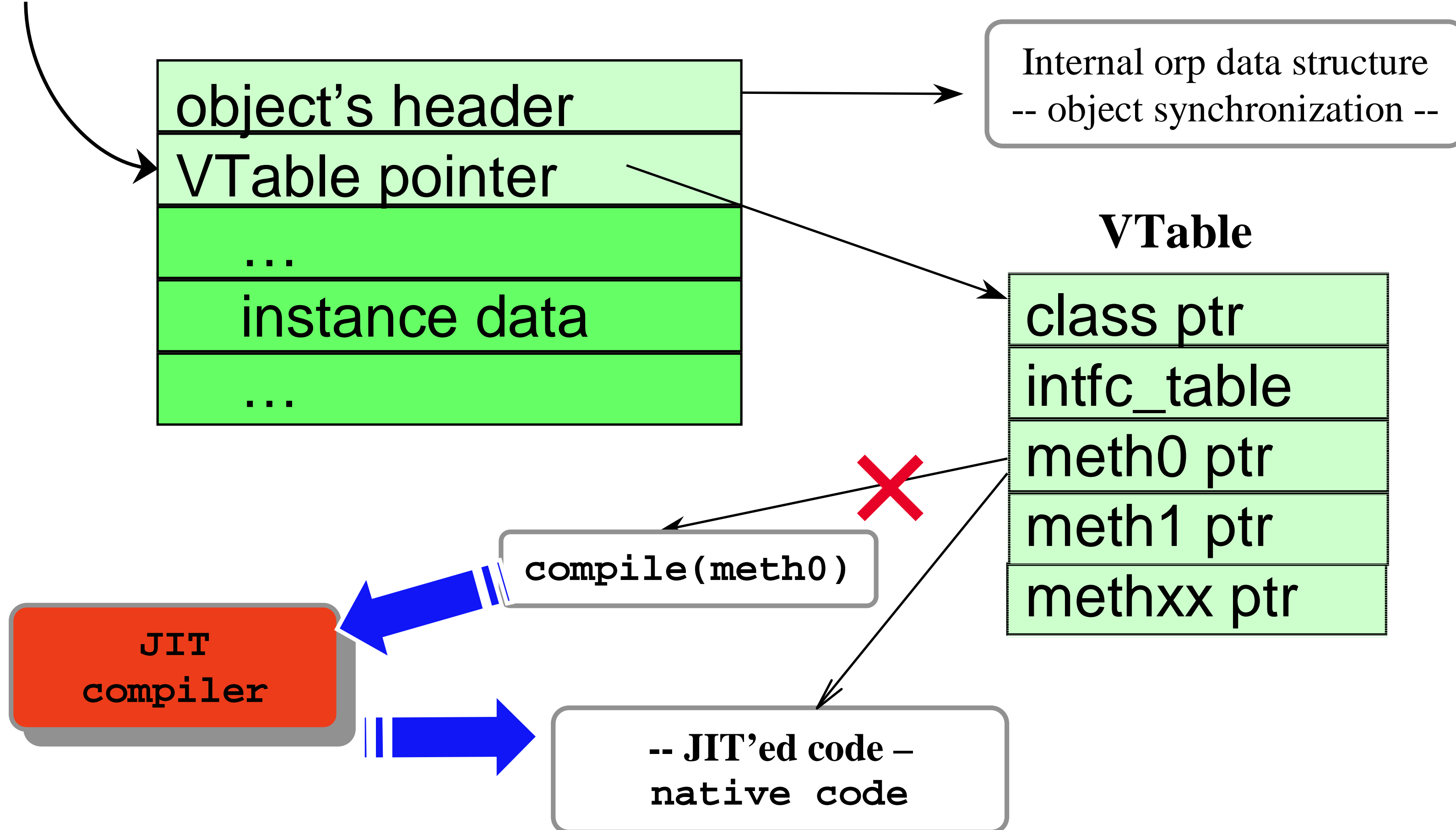
Method Data Structures



Object Layout



Object Layout



Compile-me Stub

```
mov eax, method_handle  // Method *
jmp compile_method_trampoline
```

Compile Method Trampoline

```
Create an LJF entry // will discuss later
                    // (Last Java Frame)
push method_handle  // passed in eax from
                    // compile-me stub
call jit_a_method   // returns entry
                    // point in eax
Pop the LJF entry
jmp eax             // jump to newly
                    // compiled code
```

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: use a callback 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);
```

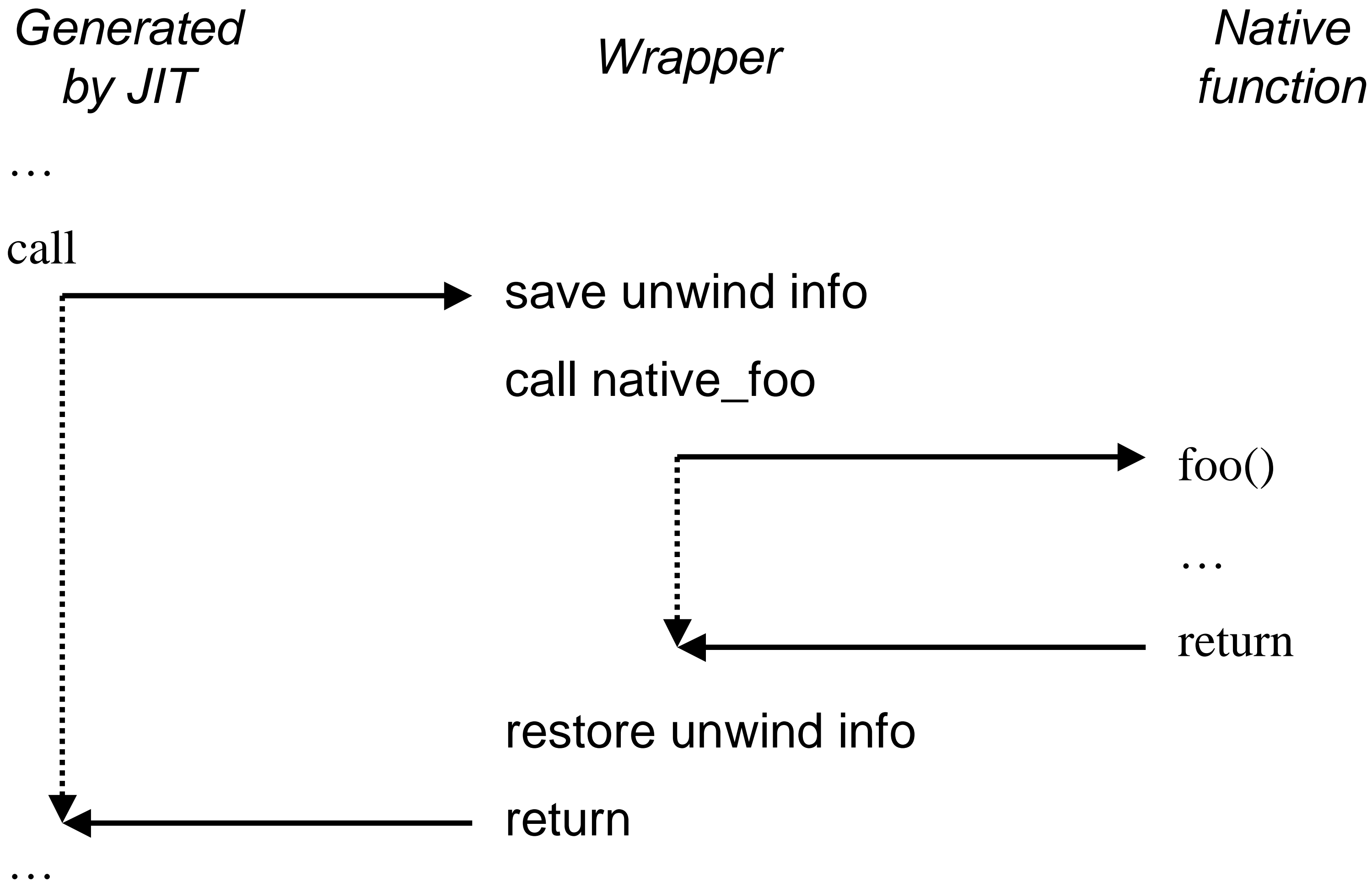
Unwinding: Main Loop

```
while(...) {
  jit_info = methods.find(*(context->p_eip));

  if(stack frame is Java) {
    jit_info->get_jit()->unwind_stack_frame
       (jit_info->get_method(), context);
  } else {
    ok = ro_unwind_native_stack_frame(context);
}
```

Stack Unwinding: Native Methods

- We assume no cooperation from the compiler used for native methods ("C").
- Unwind information is saved on every transfer from Java to C.

Native Methods Wrappers: RNI-like



RNI-like native wrapper

```
Create LJF entry
Re-push argument
Call the method
Pop LJF entry
Return
```

RNI-like native wrapper: example

```
Java:
java.lang.VMSystem.arraycopy(java.lang.Object src,
                             int srcOffset,
                             Java.lang.Object dst,
                             int dstOffset,
                             int length);
C:
java_lang_VMSystem_arraycopy(Java_java_lang_Class *,
                             Java_java_lang_Object *src,
                             int32 srcOffset,
                             Java_java_lang_Object *dst,
                             int32 dstOffset,
                             int32 length)
```

arraycopy (1)
Create LJF entry

```
push ebp
push ebx
push esi
push edi
push 0
call get_ljf_addr
push eax
push [eax]
mov esp -> [eax]
```

`arraycopy (2)`

Re-push arguments, call the method

```
push [esp+32]
push [esp+40]
push [esp+48]
push [esp+56]
push [esp+64]
push 0xd73674          // java_lang_System
call java_lang_System_arraycopy
add 24 -> esp
```

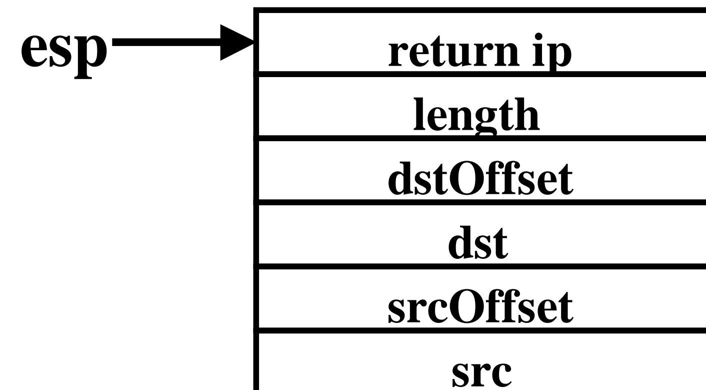
arraycopy (3)

Pop LJF entry, return

```
pop ecx
pop ebx
mov ecx -> [ebx]
add 4 -> esp
pop edi
pop esi
pop ebx
pop ebp
ret 20
```

Native stub for `arraycopy`

- Stack state at the entry to the native wrapper
- args are pushed left to right and are callee-popped

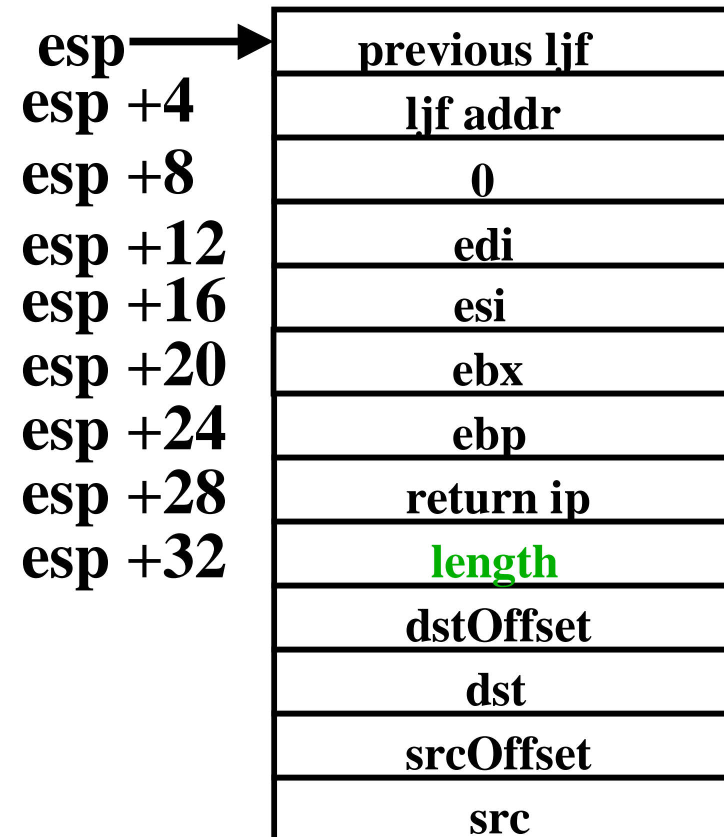


arraycopy

```
push ebp
push ebx
push esi
push edi
push 0
call get_ljf_addr
push eax
push [eax]
mov [eax], esp
```

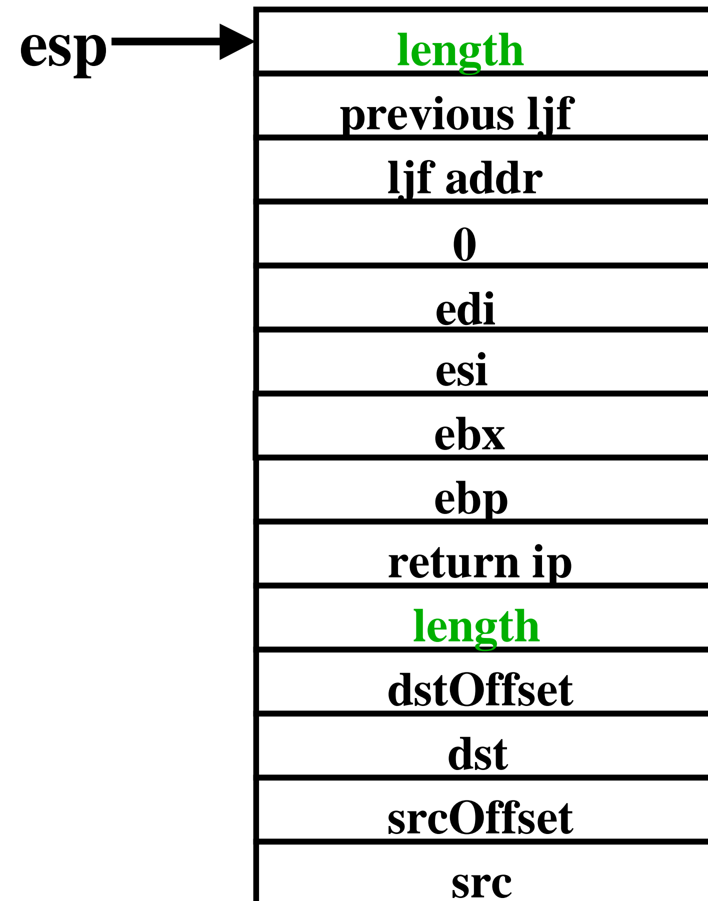
arraycopy

```
push [esp + 32]
```



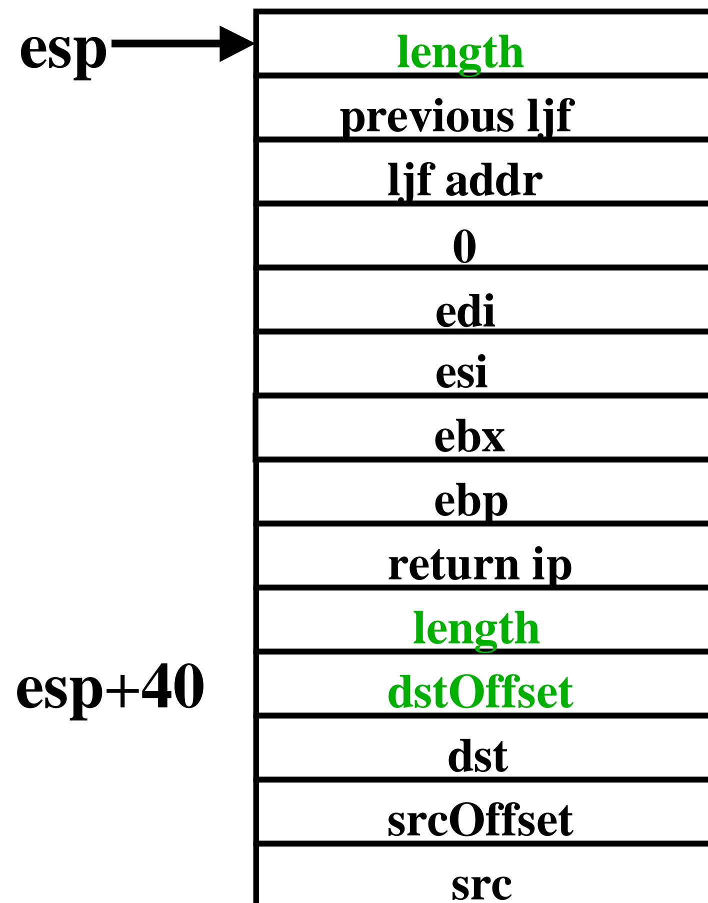
arraycopy

```
push [esp + 32]
```



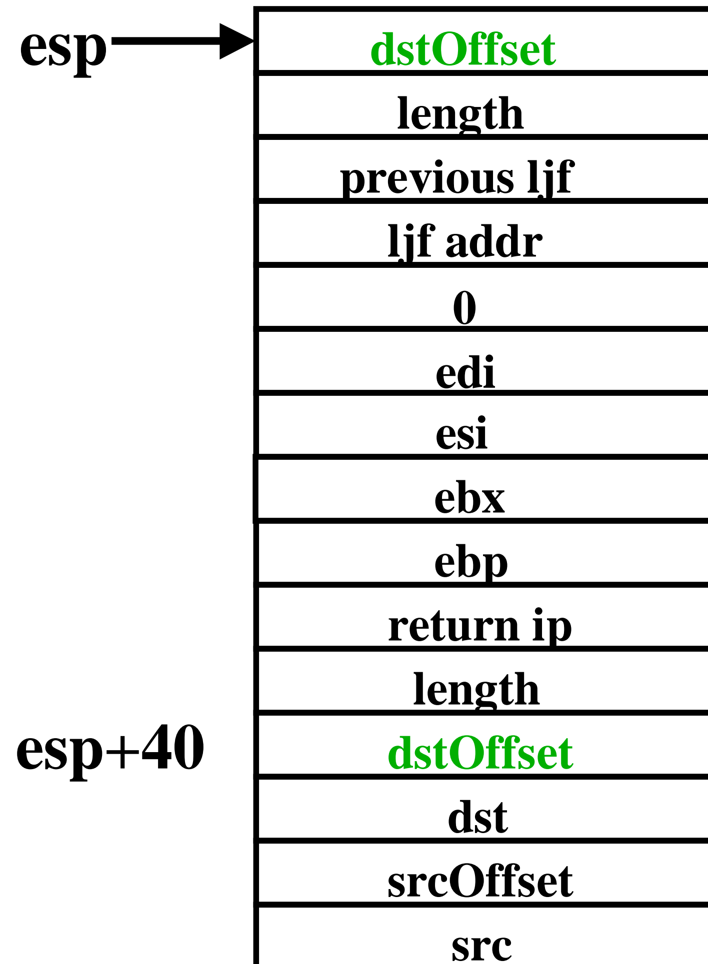
arraycopy

```
push [esp + 32]
push [esp + 40]
```



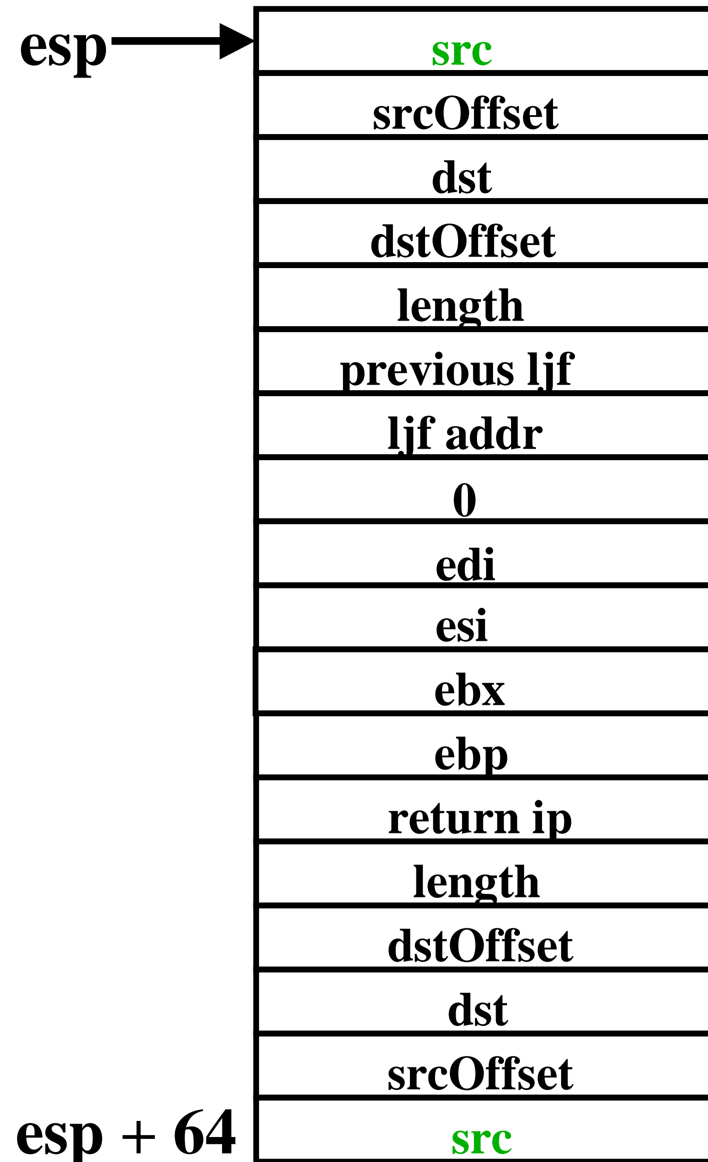
arraycopy

```
push [esp + 32]
push [esp + 40]
```



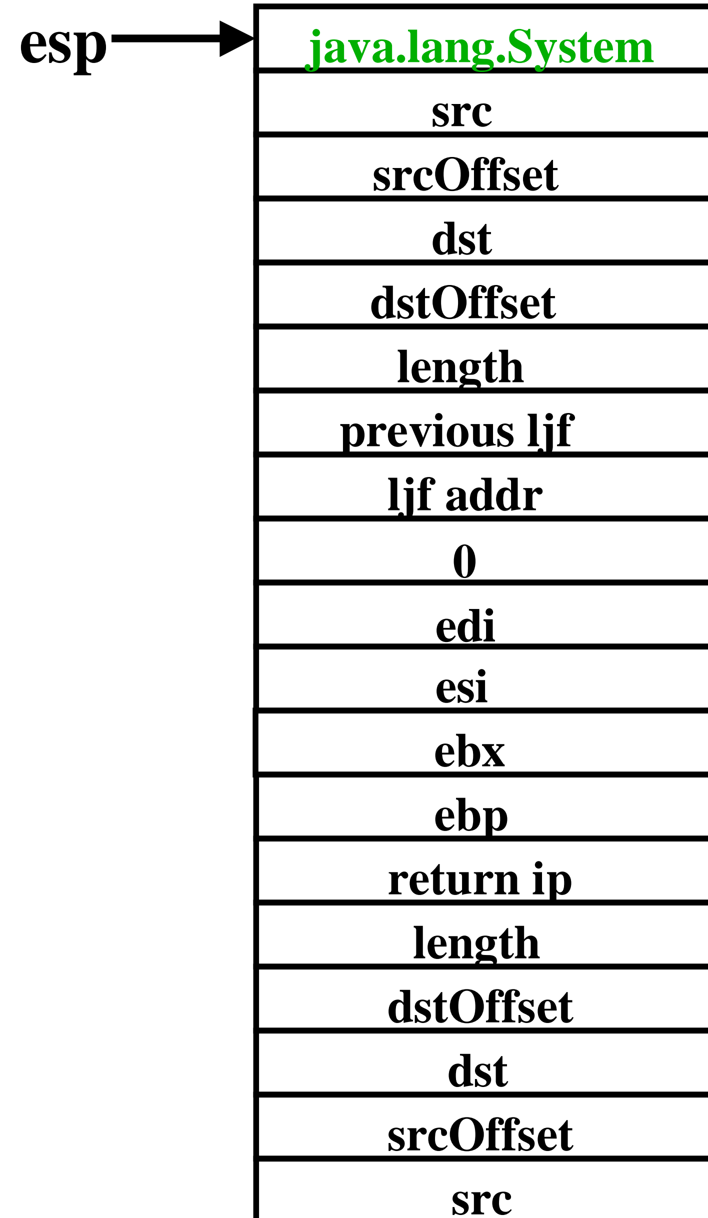
arraycopy

```
push [esp + 32]
push [esp + 40]
push [esp + 48]
push [esp + 56]
push [esp + 64]
```



arraycopy

```
push [esp + 32]
push [esp + 40]
push [esp + 48]
push [esp + 56]
push [esp + 64]
push java.lang.System
```



arraycopy

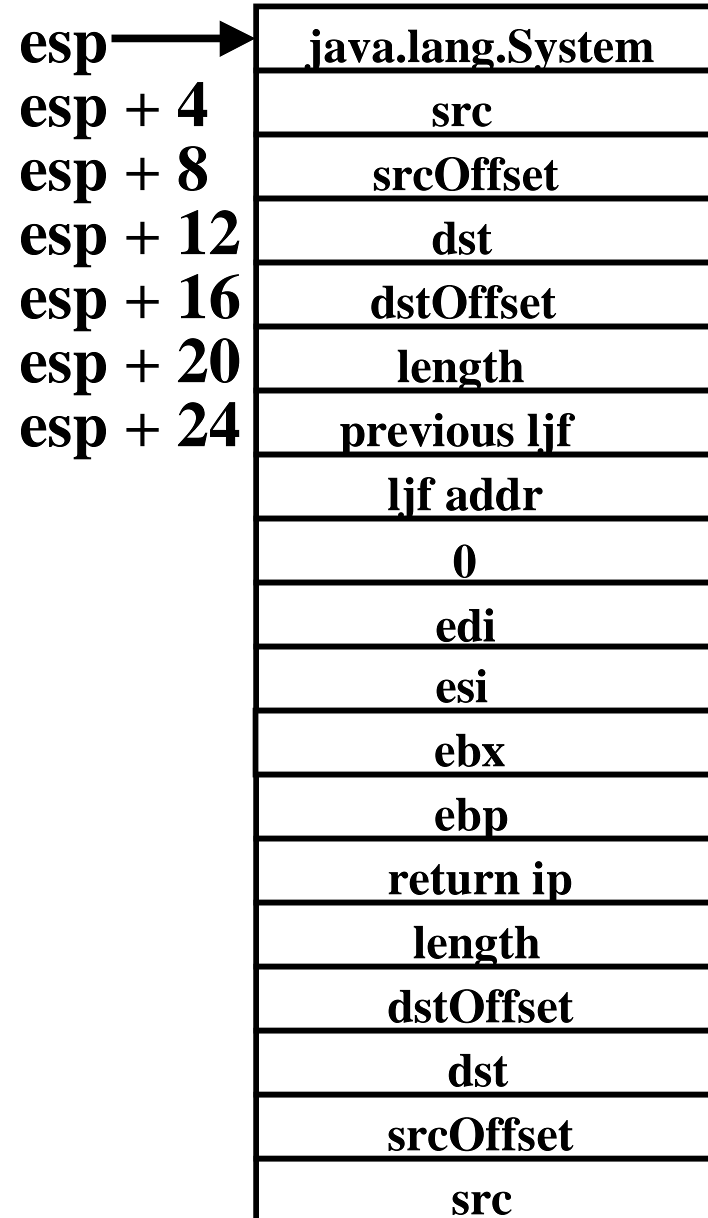
```
push [esp + 32]
push [esp + 40]
push [esp + 48]
push [esp + 56]
push [esp + 64]
push java.lang.System
call arraycopy
```

esp →

java.lang.System
src
srcOffset
dst
dstOffset
length
previous ljf
ljf addr
0
edi
esi
ebx
ebp
return ip
length
dstOffset
dst
srcOffset
src

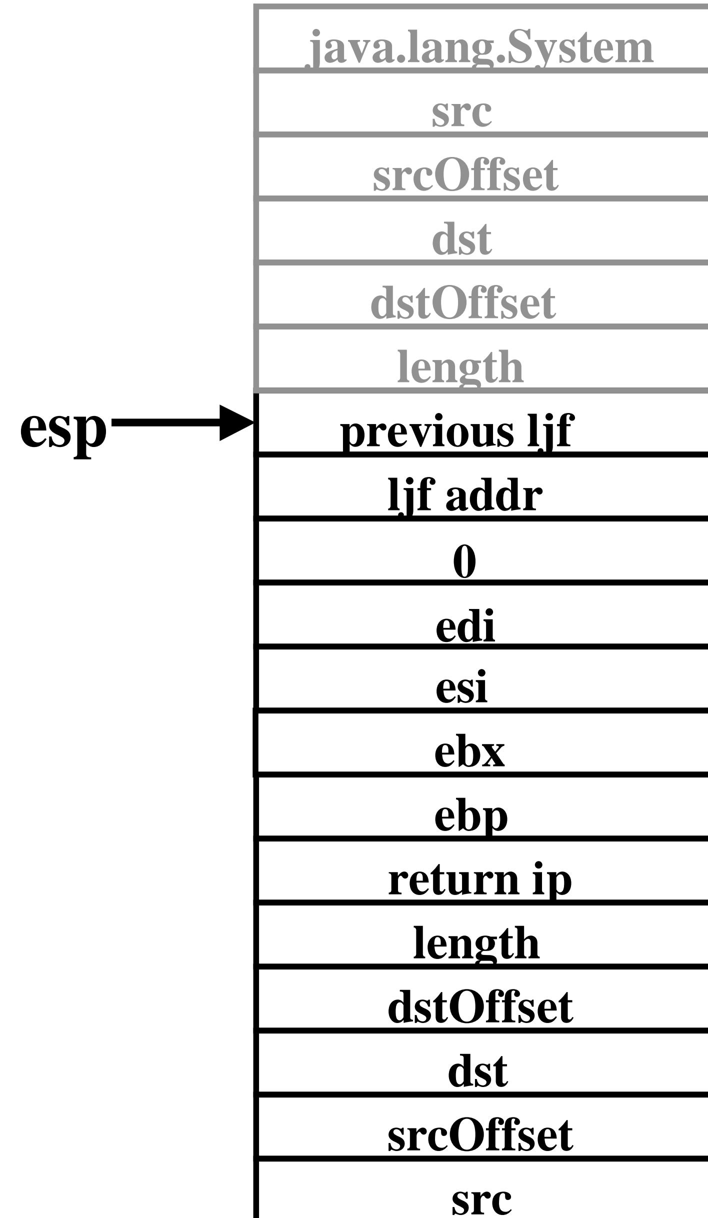
arraycopy

```
push [esp + 32]
push [esp + 40]
push [esp + 48]
push [esp + 56]
push [esp + 64]
push java.lang.System
call arraycopy
add esp, 24
```



arraycopy

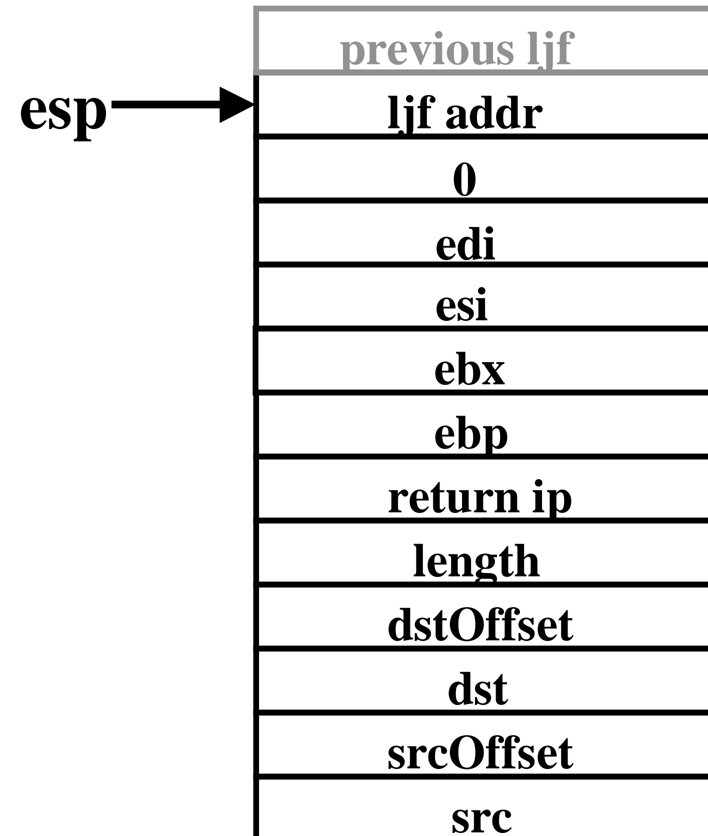
```
push [esp + 32]
push [esp + 40]
push [esp + 48]
push [esp + 56]
push [esp + 64]
push java.lang.System
call arraycopy
add esp, 24
```



arraycopy

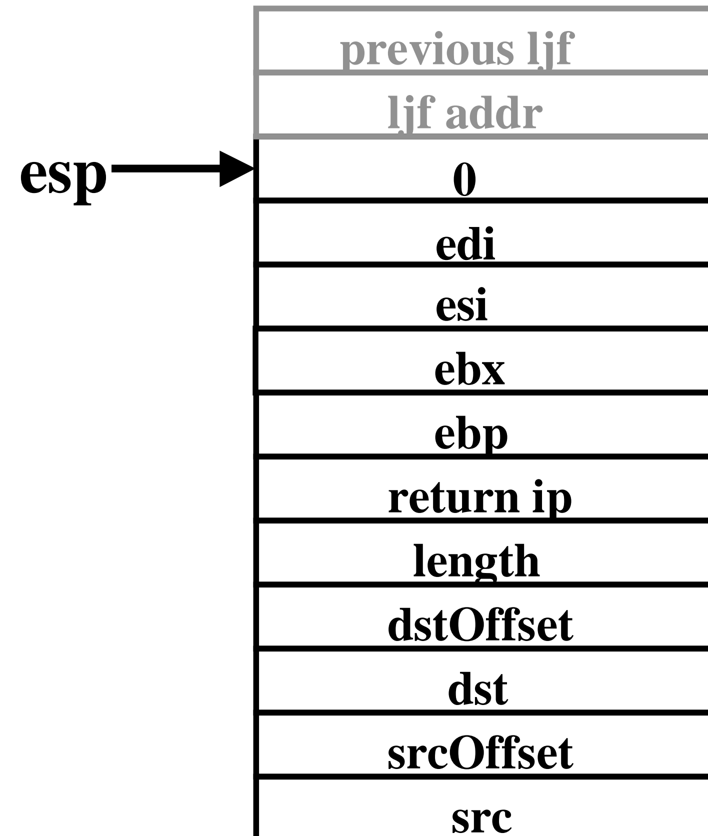
```
pop ecx
```

```
ecx == prev_ljf
```



arraycopy

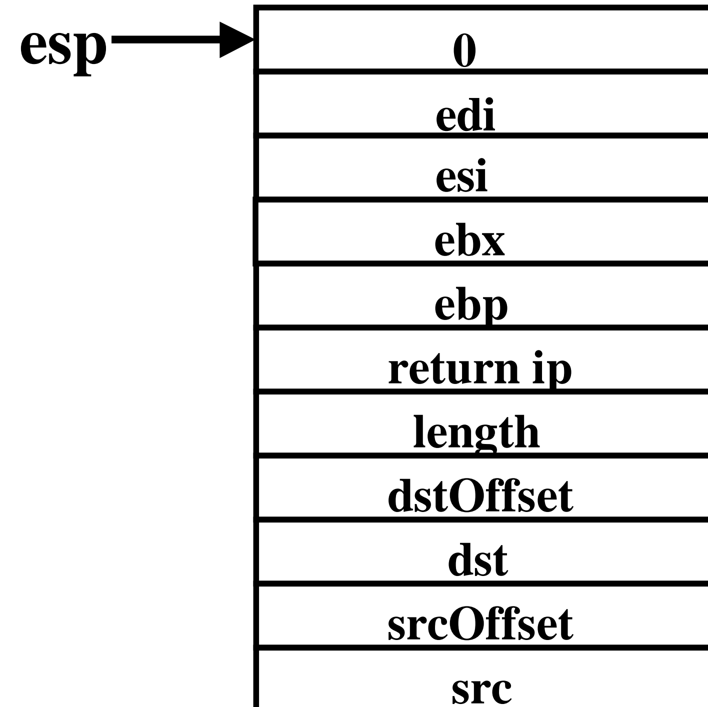
```
pop ecx
pop ebx

ecx == prev_ljf
ebx == ljf_addr
```



arraycopy

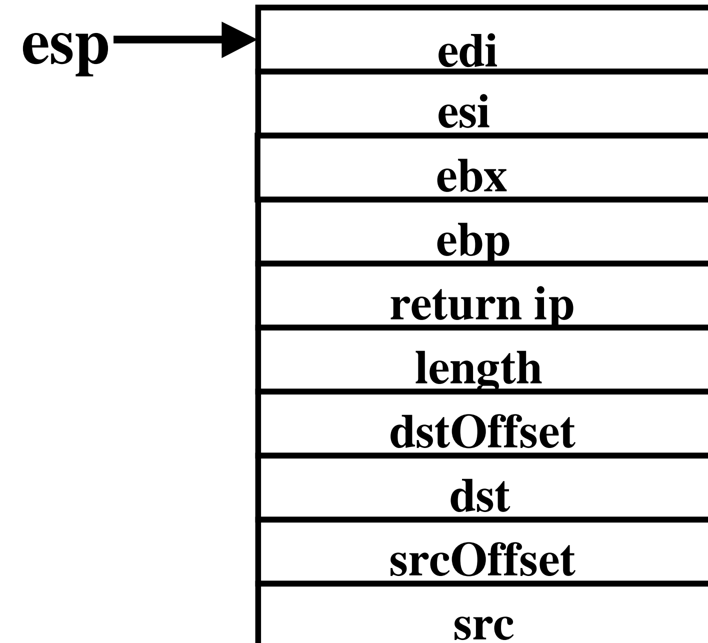
```
pop ecx
pop ebx
mov [ebx], ecx
```

```
ecx == prev_ljf
ebx == ljf_addr
*(ljf) = prev_ljf
```



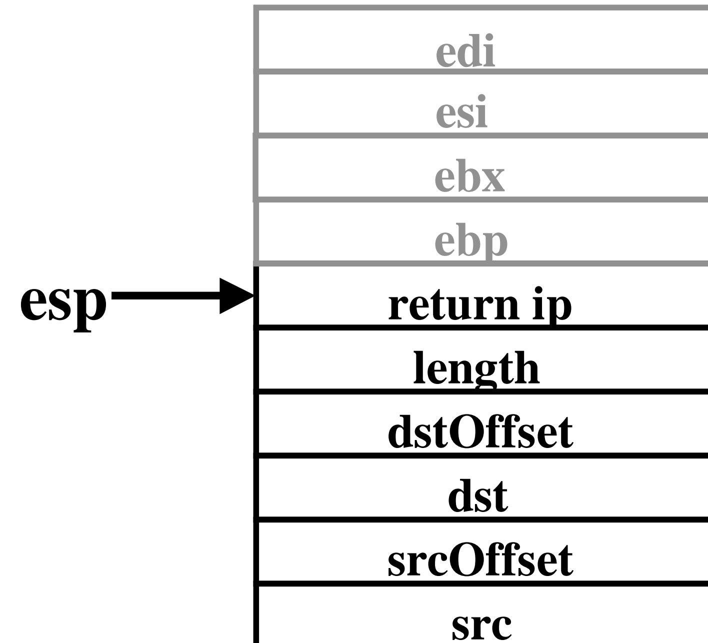
arraycopy

```
pop ecx
pop ebx
mov [ebx], ecx
add esp, 4
```



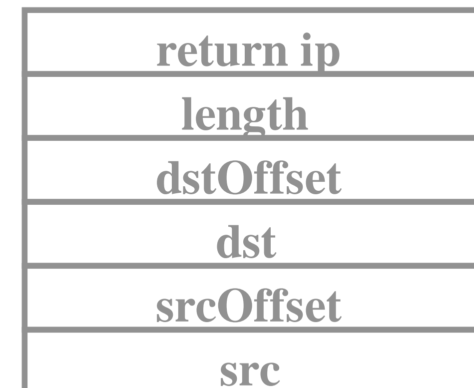
arraycopy

```
pop ecx
pop ebx
mov [ebx], ecx
add esp, 4
pop edi
pop esi
pop ebx
pop ebp
```



arraycopy

```
pop ecx
pop ebx
mov [ebx], ecx
add esp, 4
pop edi
pop esi
pop ebx
pop ebp
ret 20
```



Unwinding: Main Loop

```
while(...) {
  if(stack frame is Java) {
    // ask the JIT to unwind
  } else {
    ok = ro_unwind_native_stack_frame(context);
    if(!ok) {
      // bottom of stack
    }
}
```

Unwinding: Native Methods

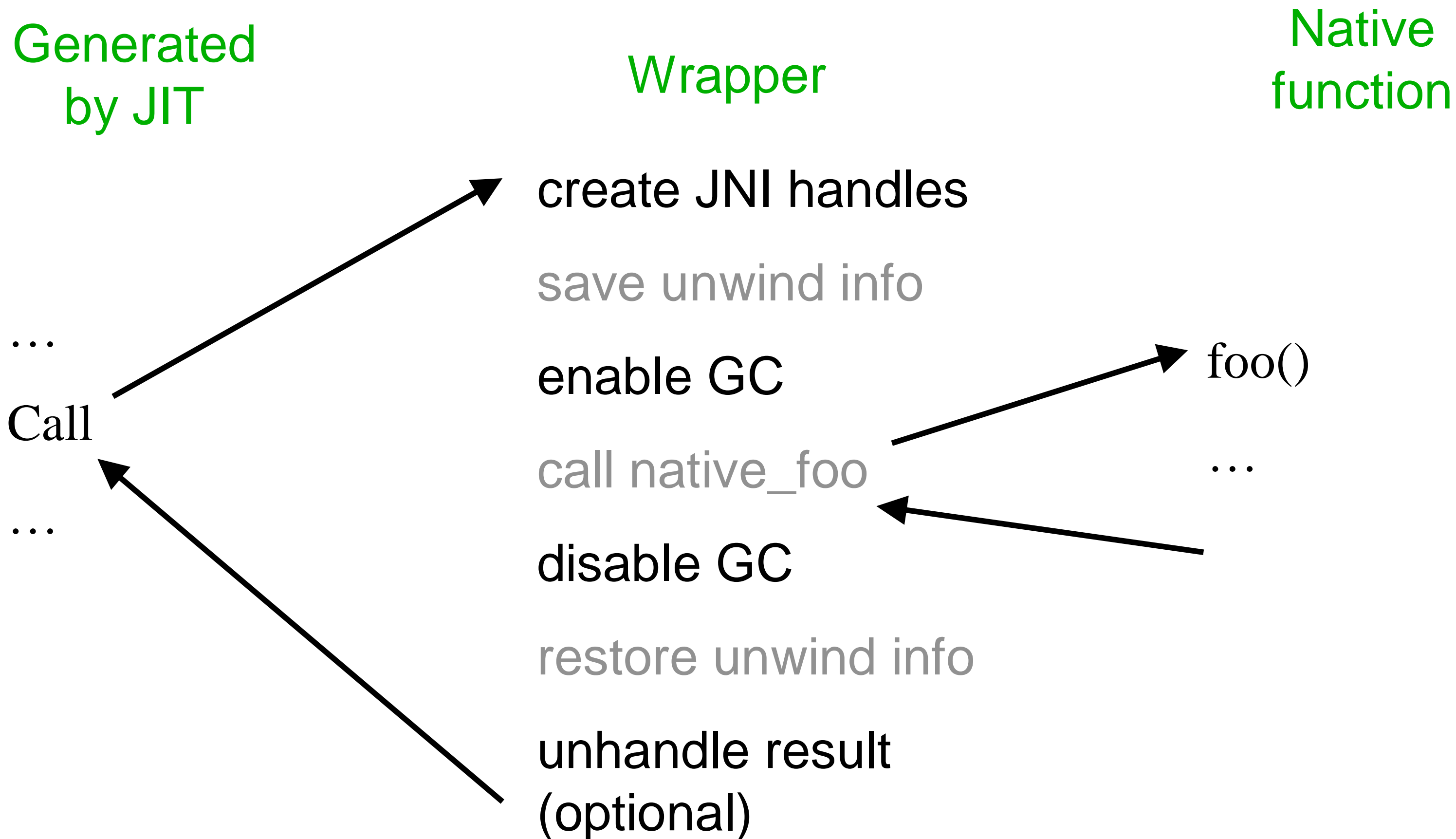
```
struct J2N_Saved_State {
    uint32          prev_ljf;
    uint32         *p_ljf;
    Object_Handle   loc_handles;
    uint32          edi;
    uint32          esi;
    uint32          ebx;
    uint32          ebp;
    uint32          eip;
}; //J2N_Saved_State
```

previous ljf
ljf addr
0
edi
esi
ebx
ebp
return ip
length
dstOffset
dst
srcOffset
src

Unwinding: Native Methods

```
ro_unwind_native_stack_frame(Frame_Context *context) {
  J2N_Saved_State *ljf = context->ljf;
  if(!ljf)
    return FALSE;

  context->ljf   = ljf->prev_ljf;
  context->p_edi = &(ljf->edi);
  context->p_esi = &(ljf->esi);
  context->p_ebx = &(ljf->ebx);
  context->p_ebp = &(ljf->ebp);
  context->p_eip = &(ljf->eip);
  context->esp   = ((uint32)context->p_eip) + 4;

  return TRUE;
} //ro_unwind_native_stack_frame
```

Native Methods Wrappers: JNI



ORP Synchronization and OS Issues
(Merging VM with OS)

intel®

Outline

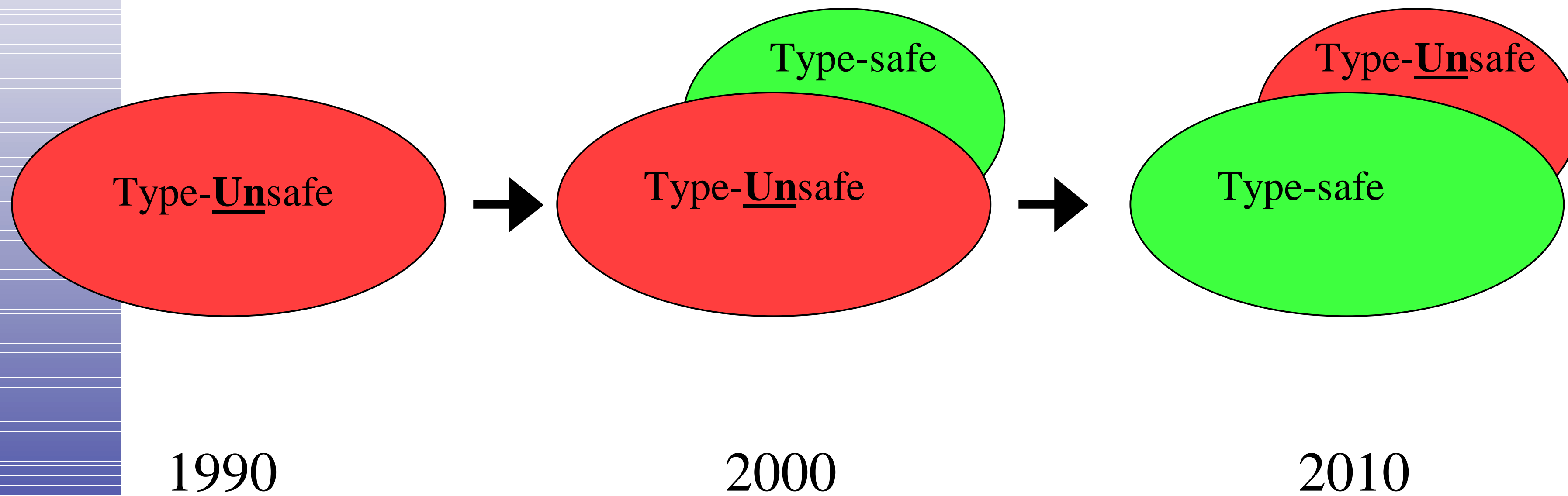
- Where VM and OS are headed
- Basic ORP design
  - Problems with existing OS APIs
- Basic ORP data structures that need tighter OS integration
  - ORP_thread
  - GC enable/disable
  - Object header bits
  - Monitorenter/exit
  - Lock_Blocks
  - Root set enumeration
- Conclusion

Where VM and OS are headed

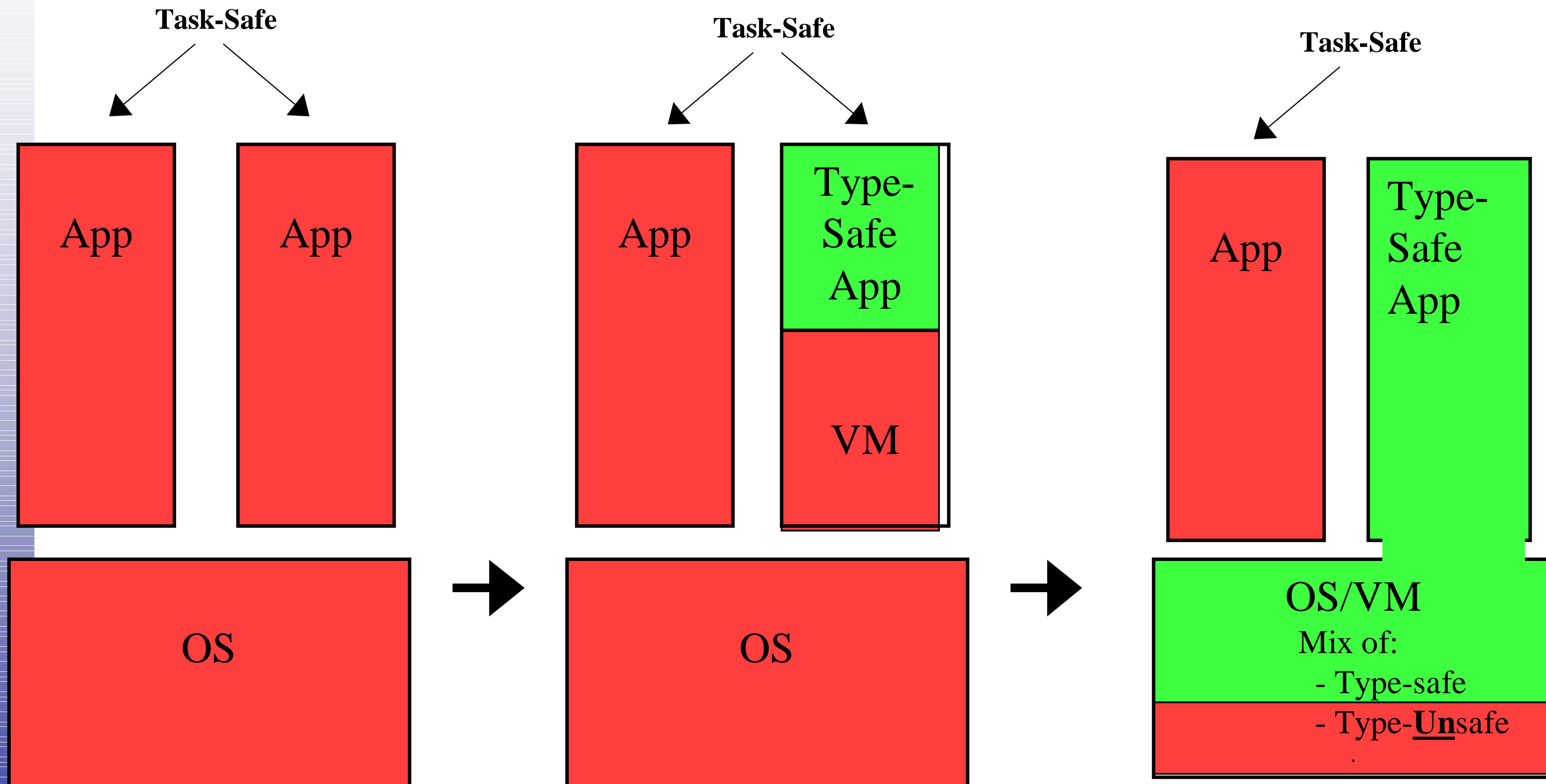
Safety

- Task-Safety
  - Provided by traditional OS
    - Isolated virtual address space protects each app
    - User/supervisor mode transition protects shared OS
- Type-safety
  - Provided by modern languages such as Java, C#
    - Programs can only address valid, accessible, compatible fields

OS Becomes Type-Safe



OS Becomes Type-Safe



Basic ORP design

Problems with exisiting OS APIs

Synch Facilities Used by ORP

Linux		**NT**
pthread_mutex	←→	SetEvent, ResetEvent
pthread_mutex	←→	EnterCriticalSection, LeaveCrit…
signals	←→	SEH (structured exception handling)
lock cmpxchg	←→	lock cmpxchg (compare and swap)

ORP Threading Model

- ORP threading is mapped "1:1:1"
- Java.lang.Thread ↔ ORP_thread ↔ OS thread
- Supports pthread (Linux) and Win2k_thread
- Problem:
  - Forced to shadow internal OS thread block inside the VM
- Really want "1:1" or even just "1"

GC Safepoint

- Definition:
  - A PC at which all live references can be enumerated
- If a thread is executing Java code
  - A PC where JIT says it is enumerable
- If a thread is executing native code
  - A PC where the thread's gc_enabled_status == enabled

GC in a Multithread Environment

- thread A tries to “new” an object but no memory available
- ORP algorithm:
  - suspend every thread at a “gc safepoint”
  - Walk every stack and enumerate the live references
- Problem:
  - Generic OS facilities are inefficient
    - Difficult to debug, lots of race/deadlock problems
- Really want tight integration with thread scheduler

Object Nurseries

- ORP has one nursery dedicated to each thread
  - Pro:
    - Avoid serialization on object allocation
  - Con:
    - if lots of threads, this is inefficient memory use

- Really want one nursery dedicated to each CPU
  - Problem:
    - Ugly unless OS internal thread scheduler is modified
  - Opportunity:
    - Research Data cache tradeoffs, thread/CPU binding tradeoffs

intel®

Memory Management

- GC
  - Focus is on recycling dead objects
  - Not worried about working set size
  - Not worried about memory quotas among threads
    - One thread can hog all the memory, starve the others
- OS
  - Basically a fully associative cache for files on disk
  - Really good at task balancing, sharing of physical RAM
  - No concept of live/dead objects
- Problems
  - OS and GC are mutually unaware of each other's policies
  - Do they conflict? If so, how?
  - Can a unified GC-OS memory management model be constructed?

Basic ORP Data Structures That Need Tighter OS Integration

- ORP_thread
- GC enable/disable
- Object header bits
- Monenter/monexit
- Lock_Blocks
- Root set enumeration

ORP_thread Data Structure

```
class ORP_thread {

    Java_java_lang_Thread   *p_java_lang_thread;
    Java_java_lang_Object   *p_current_object;
    Java_java_lang_Object   *p_exception_object;

    ORP_thread              *p_free;
    ORP_thread              *p_active;

    Lock_Block              *p_free_lock_blocks;
    Lock_Block              *p_active_lock_blocks;  // used by 'L' command in debugger

    POINTER_SIZE_INT        quick_thread_index_shifted_left_with_recursion_set_to_one;

    java_state              app_status;
    gc_state                gc_status;

    bool                    interrupt_a_waiting_thread;
    bool                    thread_is_java_suspended;

    Registers               regs;
```

ORP_thread Data Structure
(continued)

```
J2N_Saved_State *      last_java_frame;

void                   *p_nursery;

// For stack trace creation
Frame_Context *throw_context;
Boolean       throw_context_is_first;

// RNI-style GC approach for native methods
gc_enable_disable_state gc_enabled_status;
GC_Frame               *gc_frames;


// gc enumeration support

bool restore_context_after_gc;
Frame_Context gc_frame_context;
trap_state which_trap;
```

GC enable/disable Design

- No C compiler support for live references
- Critical transition
  - Branch from Java method to native method is the critical transition
- Protecting the transition from Java to native code
  - By design, gc_enabled_status is always disabled except in native code
  - gc_enabled_status only set to enabled once live references are stored
- Optimization: enabled_will_block
  - Long running native methods are not suspended during GC cycle

GC enable/disable Design

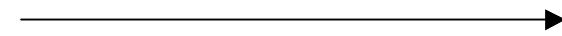
- Valid state changes

Java ←→ native disable ←→ native enable ←→ native enable will block

GC enable/disable Design

Java code

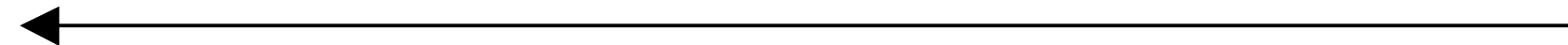
-- -
Push a live reference on the stack
Call a native method



Native C code

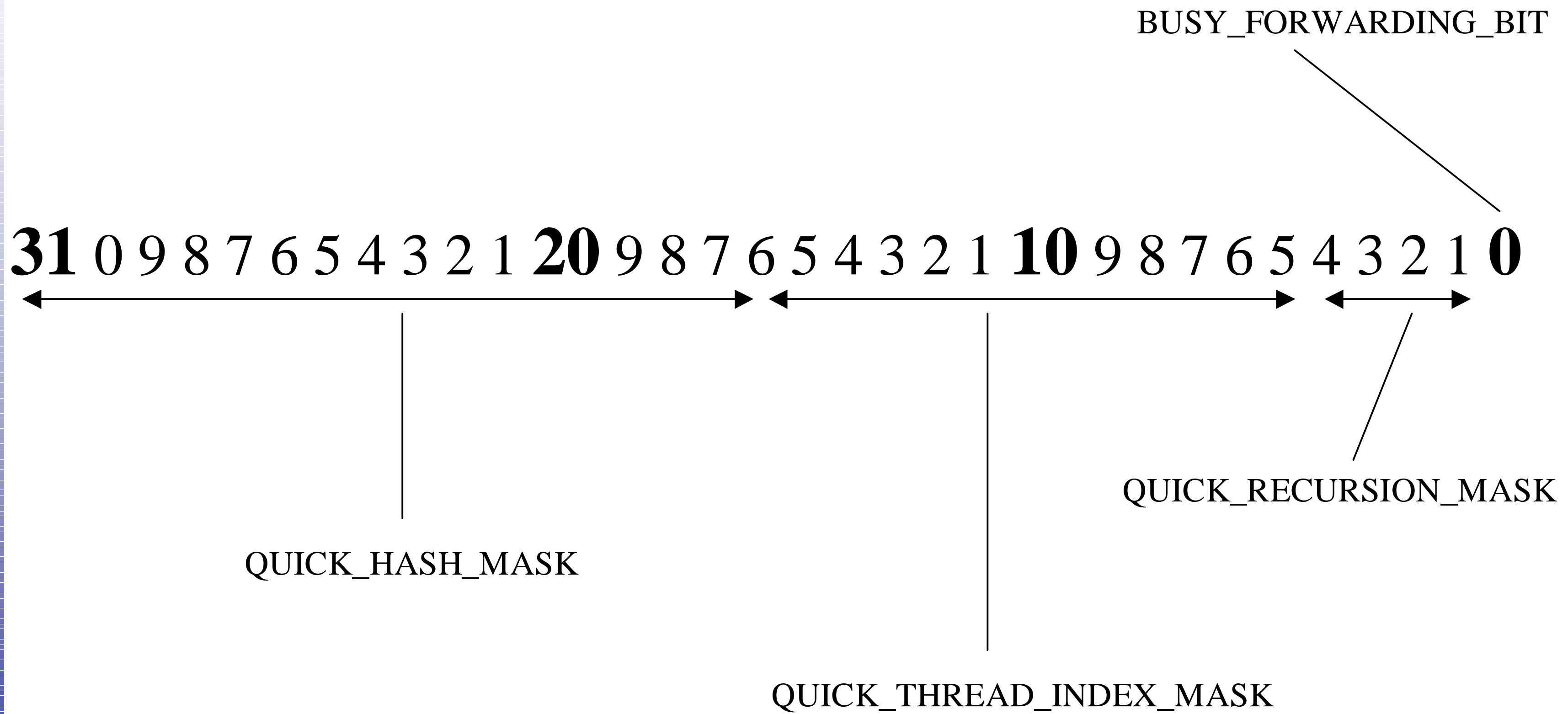
save the live reference
---
enable
-----
enable_will_block

-- some long running native app
---
enable
disable
return



-
-- --

Object Header Bits



monitorenter

Monitorenter bytecode calls the following JVM internal function:

```
Boolean orp_monitor_enter_or_null(Java_java_lang_Object *p_obj)
{
   int *p_header = p_obj;

   p_header--;  //  lock header is at offset -1 word;

   if  (  InterlockedCompareExchangePointer (  p_header,
                                               quick_thread_index_shifted_left_with_recursion_set_to_one,
                                                UNCONTESTED_HEADER_VALUE  )

             = = UNCONTESTED_HEADER_VALUE  )

                 return TRUE; // this is the common case


   otherwise return FALSE;  // caller will then do a more complex, slower algorithm that will manipulate
                            // Lock_Blocks


}
```

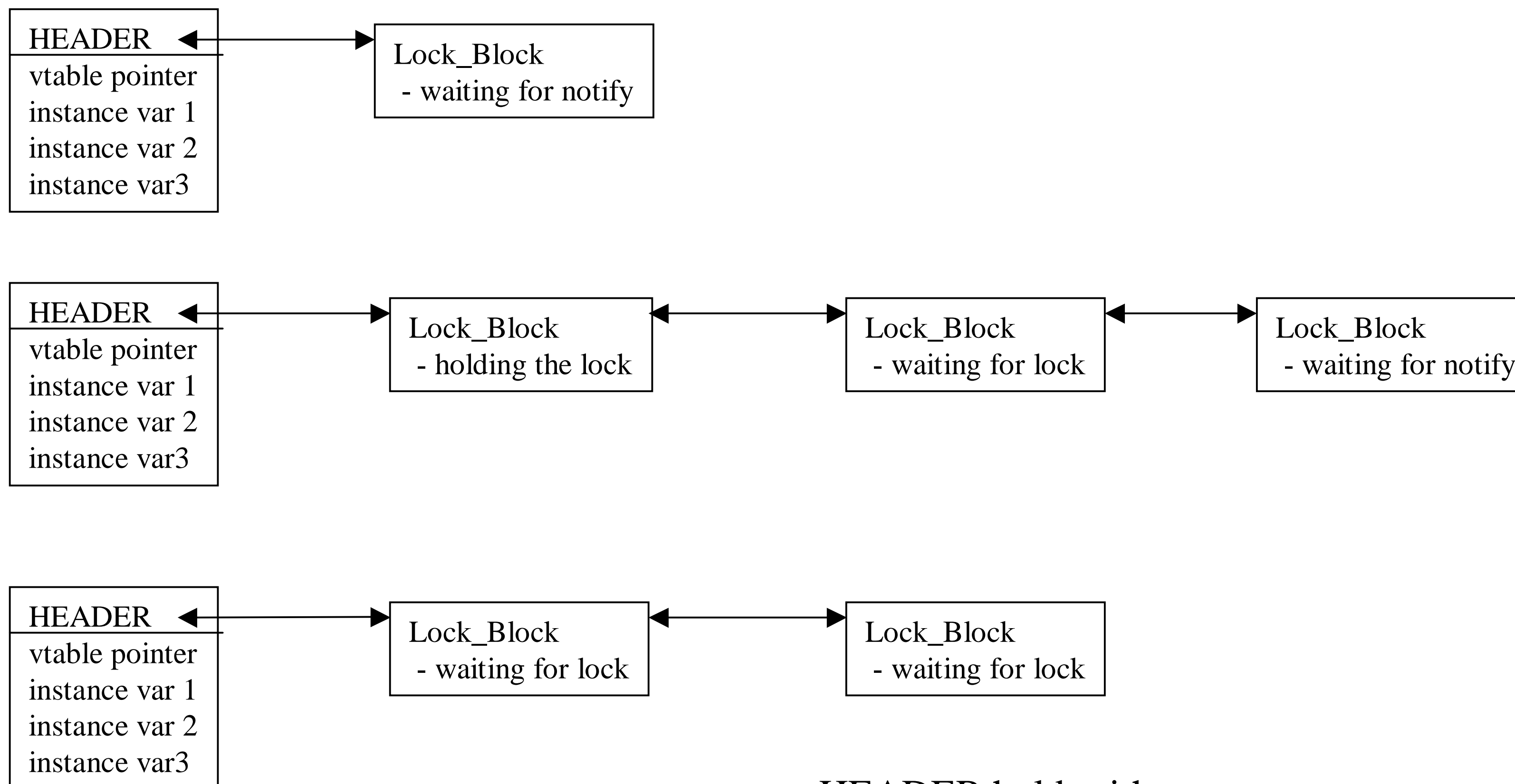
monitorexit

Monitorexit bytecode calls the following JVM internal function:

```
Boolean orp_monitor_exit_or_null(Java_java_lang_Object *p_obj)
{
   int *p_header = p_obj;

   p_header--;  // hash forward lock header is at offset –1 word;

   if  (  InterlockedCompareExchangePointer (  p_header,
                                               UNCONTESTED_HEADER_VALUE,
                                               quick_thread_index_shifted_left_with_recursion_set_to_one)

          = =  quick_thread_index_shifted_left_with_recursion_set_to_one  )

                          return TRUE;  // this is the common case


   otherwise return FALSE;  // caller will then do a more complex, slower algorithm that will manipulate
                            // Lock_Blocks


}
```

Internal monenter/exit Data Structures

JAVA OBJECTS



HEADER holds either:

1) on demand, the hash and/or lock bits
2) GC forwarding pointer
3) Lock_Block pointer

Root Set Enumeration Traps

```
enum trap_state {
    x_the_safepoint_control_thread,
    x_lock_enum,
    x_orp_enable_gc,
    x_suspend_in_java_frame,
    x_java_suspended,
    x_at_breakpoint,
    x_orp_stop_thread_for_gc_ret,
    x_orp_throw,
    x_orp_execute_java_method_array,
    x_java_sleep,
    x_java_wait,
};
```

To enumerate, thread must be in one of the above states

Root Set Enumeration Design

SUSPENDING THREADS AND ENUMERATING LIVE REFERENCES

```
Orp_enumerate_root_set_all_threads()
{
  p_the_safepoint_control_thread = current thread   ← make the current thread the owner of this GC cycle

  // NOTE:  the java app debugger requires all threads to get suspended at gc safepoint when breakpoint is hit
  call enum_when_no_debugger()

      for each thread

          // threads suspended by java.lang.Thread.suspend(), java.lang.Thread.sleep() or java.lang.Object.wait
          // are by design, left in the enumerable state

          if (java_suspended)
              orp_enum();

          else if (thread_is_sleeping or thread_is_waiting)
              move this thread's state from Enabled to EnabledWillBlock
              orp_enum();

          else
              // we have to let the thread fall into an enumeration "trap"

              thread->gc_status = gc_moving_to_safepoint
              ResetEvent(thread->gc_resume_event_handle);  // thread will eventually block on this event

              while (1)   {
                 sleep(2);  ← let the thread run for 2 milliseconds
                 if (thread has fallen into a trap)
                    orp_enum();
              }
```

Root Set Enumeration Design

RESUMING ALL THREADS AFTER GC IS DONE

```
For each thread {

    if (restore_context_after_gc)
        thread_gc_set_context();      ← JIT may have live references in hardware registers that need updating

    else if (which_trap = = java_sleep or java_wait)
        move gc_enabled from Enabled Will Block to Enabled

    else if (which_trap = = thread_is_java_suspended)
        basically do nothing except fix up some thread state variables

    SetEvent(thread->gc_resume_event_handle)    ← turn off the which trap event

}
```

VM/OS Research Opportunities

- Memory management
- Threading
- Synchronization
- TLB organization
  - dirty bit support for GC pointer tracking
- User/supervisor mode – who needs it??
- Multiple virtual address spaces – who needs it??

Conclusion

- Traditional OS never designed for VM running as an app
- Many fundamental design issues need to be revisited
- Need an open source VM/OS research platform
  - Starting point: Linux/ORP

Garbage Collection in ORP

intel®

Approach

- Toolkit comprised of several algorithms
- Defined interface between VM, JIT and GC
- Starts with train algorithm
- Use train to mimic most other algorithms
- Provide interface to support concurrency

Blocks

- Divide heap into power of 2 blocks
- Block tables indexed by shifted address
  - Generation and Train/Car/Step/Nursery id
  - Other bookkeeping info such as block end
- Multiple contiguous blocks for nursery
  - Used for nurseries
  - Used for large fixed objects

Fixed Object Space

- For large objects that are too expensive to move
  - Bit maps
  - Popular objects
- For class data structures, to simplify the JIT
- Fragmentation (so far) not a problem
- Circular First Fit algorithm
  - Good tradeoff in allocation time vs. fragmentation
  - Could be better
    - Need to build and measure a bibop scheme

Eviction and Scavenging

- Object oriented
- One evicts an object from its current space
- Using table we locate which space to move object
- Target space does the move
- All table and OO driven so code is clean

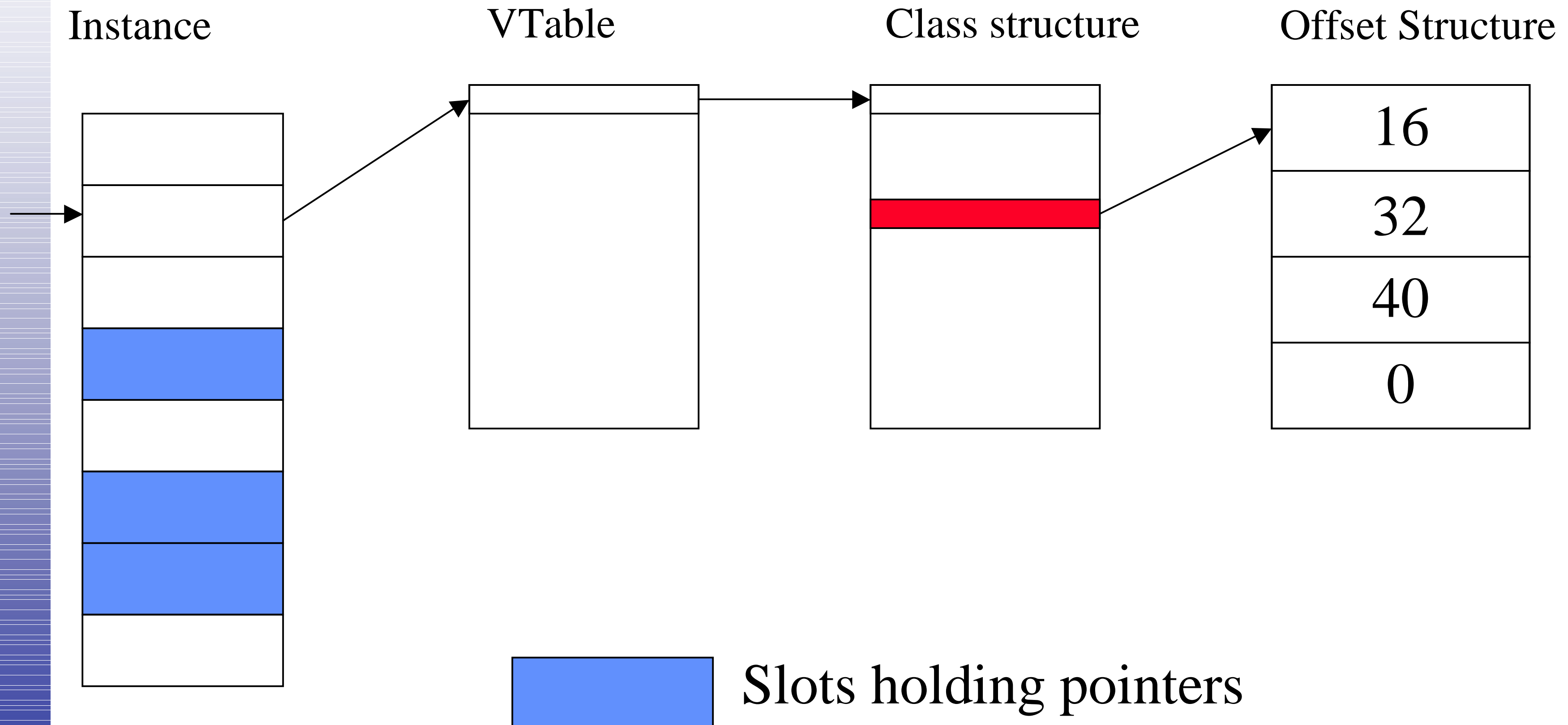
Nursery and Step (YOS)

- Currently one nursery per thread
  - Various nursery per CPU proposals floating around
- Nursery and steps collected together at each GC
  - No write barrier overhead for slots in nursery and step
- Steps ages objects
- Can move entire blocks from nursery to step
  - Useful when there is lots of free space

Cars and Trains (MOS)

- Mature Object Space divided into cars
- Cars made up of blocks and associated with train
- Oldest step is scavenged into youngest car in youngest train
- Do one car plus YOS at each collection

Pointer scan structure



ORP Compile Time Defines

- GC_FIXED_V1
  - Makes all objects fit into LOS
  - Set threshold so all objects are considered large
  - No nurseries
  - Multiple Large Object Spaces
- GC_COPY_V2
  - Just nursery and 1 Step that pours into itself
  - Nursery and Step pour into Step
- GC_GEN_V3
  - Nursery + Step generation
  - 2 Trains of 1 very large car each – objects poured back and forth

(More) ORP Compile Time Defines

- GC_TRAIN_V5
  - Multiple trains with multiple cars
  - Nursery, Steps, Large Objects all used
  - Collects Nursery and Step and one Car each cycle
- GC_SAPPHIRE
  - Requires JIT_SAPPHIRE
  - Provides concurrent GC
  - (see paper in JavaGrande ‘01)

Read/Write Barriers

- GC_GEN_V3
  - Uses card marking and call interface
  - -jito3 fastwb inlines card marking
- GC_SAPPHIRE_V5
  - Provides call write barrier interface for all writes to the heap
  - Working on providing call read barrier interface
- Linux does not provide fast interface to page dirty bits so compiler provides write barrier
- Use the read and write barriers to generate complete heap trace

Concurrent Tri-color Mark and Sweep

- Start with GC_FIXED_V1
- Turn on write barrier calls using gc_requires_barriers routine
- Redo write barrier gc_heap_write_ref to enforce no black to white invariant
- Presto we have a Dikstra style tri-color algorithm
- I actually did this as part of debugging Sapphire

Your Algorithm Here

- Understand how current algorithms work so you don't redo a lot of work.
- Define a new GC_MY_COOL_ALGORITHM_V6
- Insert your code using only pieces that are different
- Build and measure against the other tuned algorithms
- ORP allows apple to apple comparisons
- Expose, explore, and exploit

Debugging Your New Algorithm

- gc_trace
  - Takes an object address and a string
  - If object is distinguished the string is printed along with the object
- Add calls to gc_trace throughout your new code so you can follow bogus objects through their life to determine what goes wrong
- Turned completely off in release mode

GC Plan file

- gc_plan.cpp
- Provides lots of knobs for adjusting size of the various gc strucutures
- Key to painlessly changing configurations

Torturing Your New Algorithm

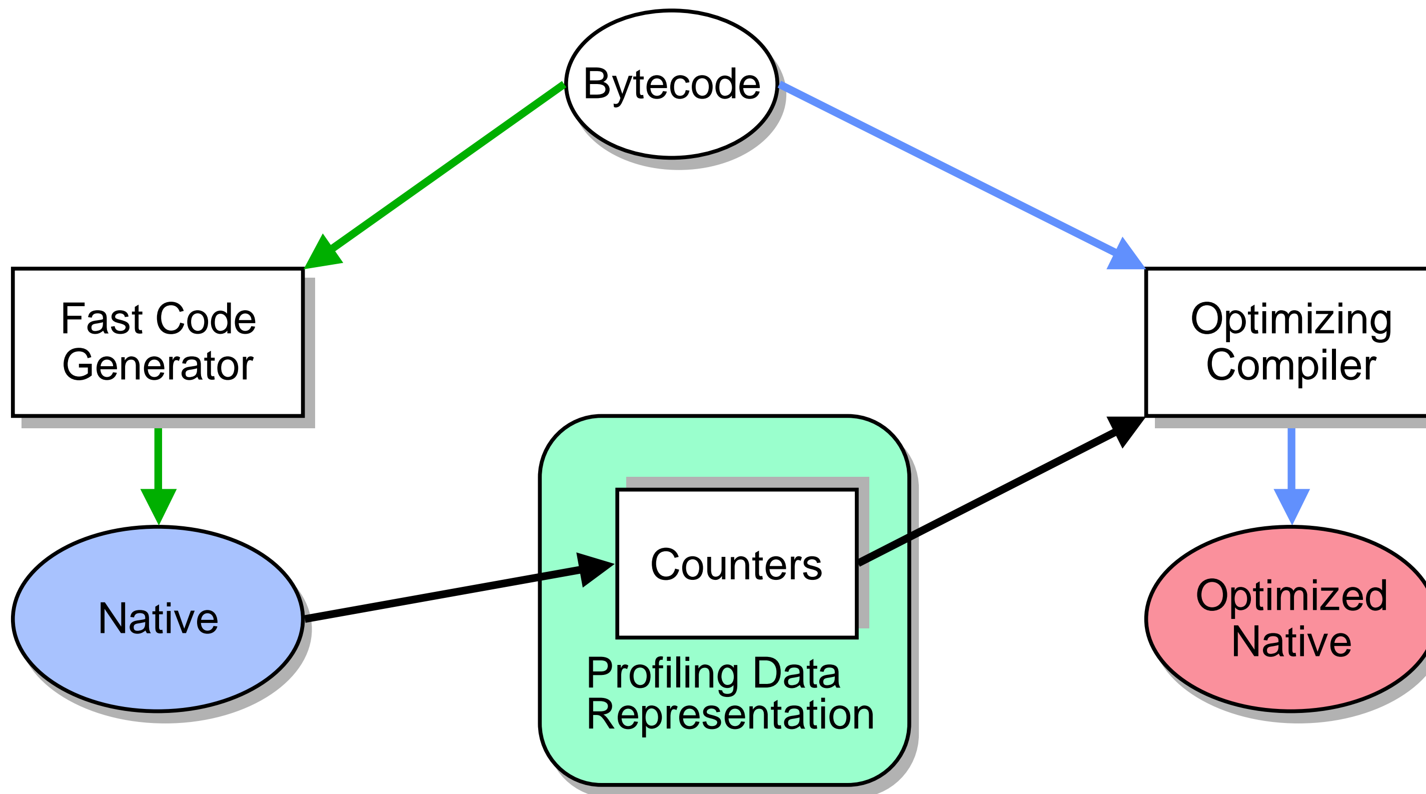
- GC torture test provide Java based framework to stress test the GC
- Some stress write barriers, others allocation
- Some run lots of GC in one frame and allocation or mutators in other threads
- Provided so you can quickly generate your own torture tests

Conclusions

- Platform for investigating new GC algorithms
- Easy to mimic most known algorithms
- The key is that we started with a full blown train algorithm with nurseries and a fixed object space
- We added concurrency
- We are adding read barriers

Just-In-Time Compilation

Structure of dynamic recompilation



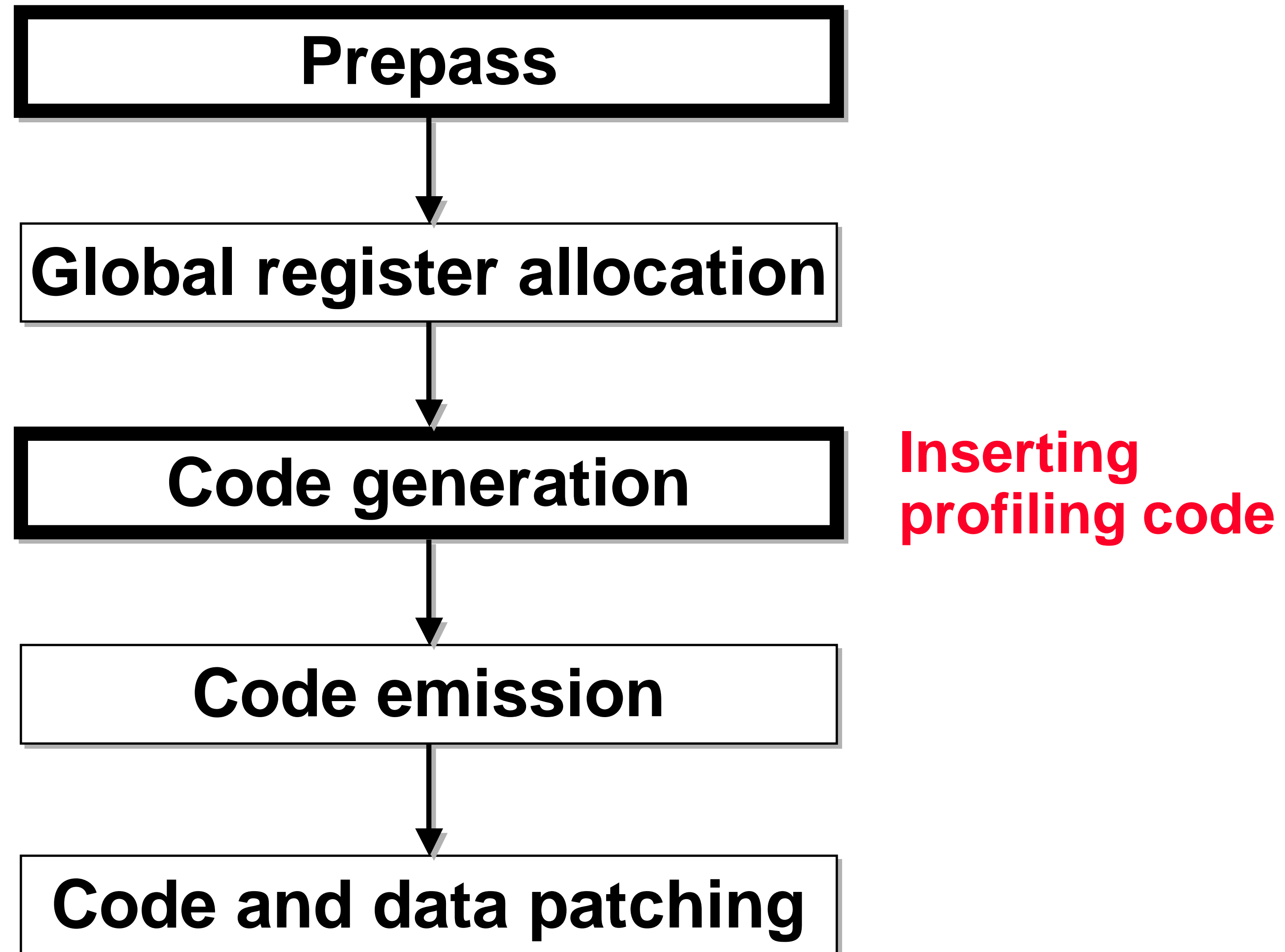
Outline

- Fast code generation
- Optimizing compiler
- Recompilation
- Usage

Fast code generator

- Fast code generation
  - 2 linear-time passes over bytecodes
- No explicit IR
  - No instruction list
  - No control flow graph
    - Except for global register allocation
- Fast global register allocation
  - Priority-based
  - No interference graph

Structure of fast code generator



Lightweight optimizations

- Lazy code selection
  - Fold memory and immediate operands
- Common subexpression elimination
  - Compare bytecode strings
- Array bounds checking elimination
- FP optimization (FP stack)
- Priority-based register allocation
- Load-after-store elimination
- Out-of-line exception throws
- Strength reduction

intel®

Lazy code selection

- Single-pass code generation strategy
- Folds operands lazily into compute instruction
  - Takes advantage of IA32’s rich addressing modes
- Delays generating code for memory and immediate operands
  - Tracks Java operand stack values with mimic stack

Example: z = x + 1

```
iload x

iconst_1

iadd


istore z
```

Example: z = x + 1

iload x

iconst_1

iadd

istore z

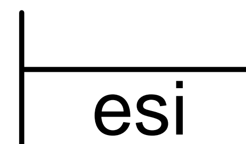
Example: z = x + 1

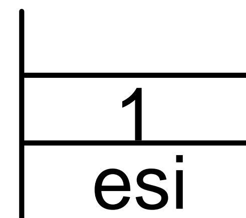
iload x

iconst_1

iadd

istore z

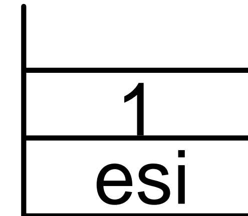




Example: z = x + 1

iload x

iconst_1



iadd

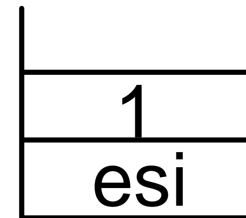
```
mov  eax, esi
add  eax, 1
```

istore z

Example: z = x + 1

iload x

iconst_1



iadd

```
mov  eax, esi
add  eax, 1
```

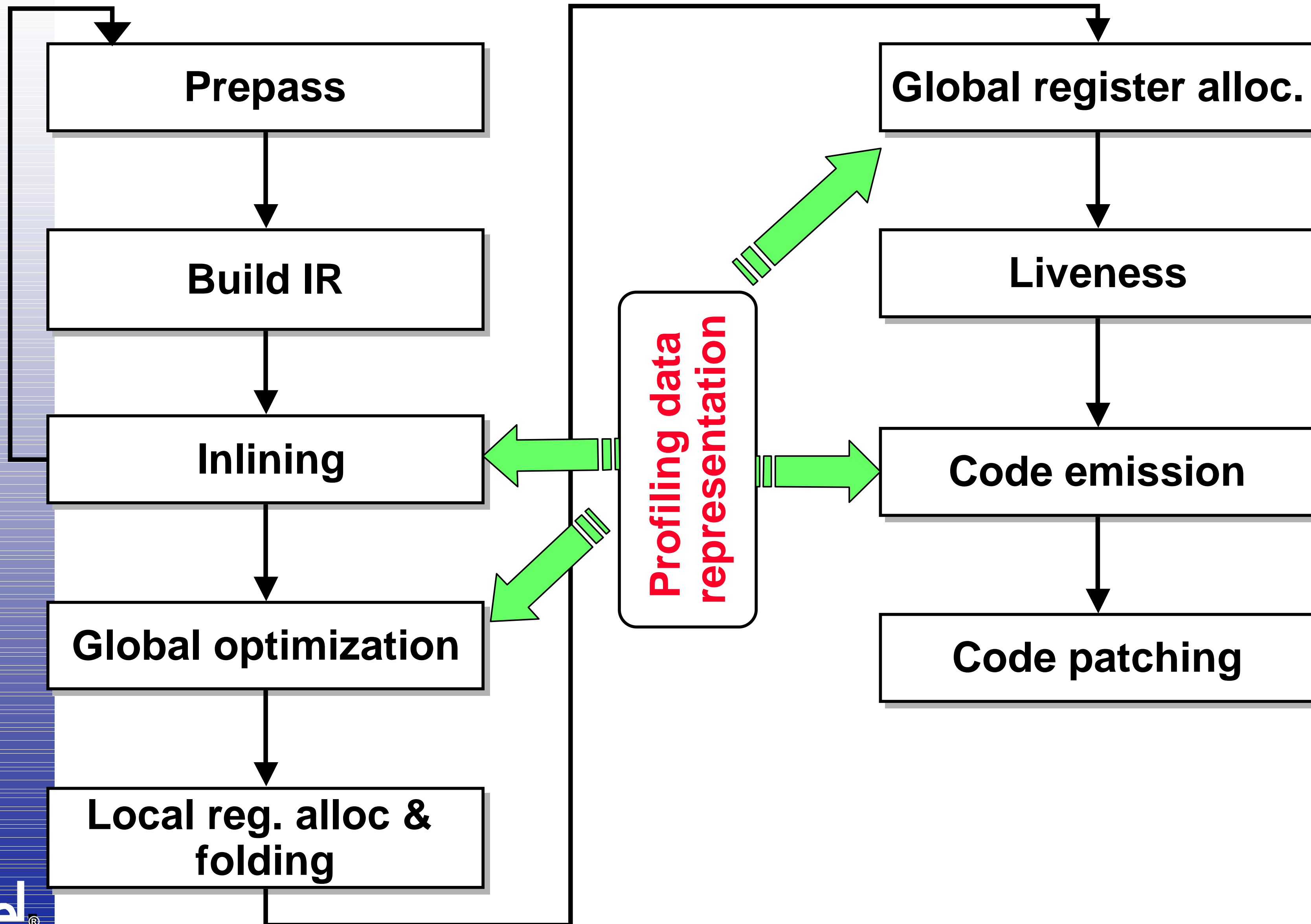
istore z

```
mov  z[esp], eax
```

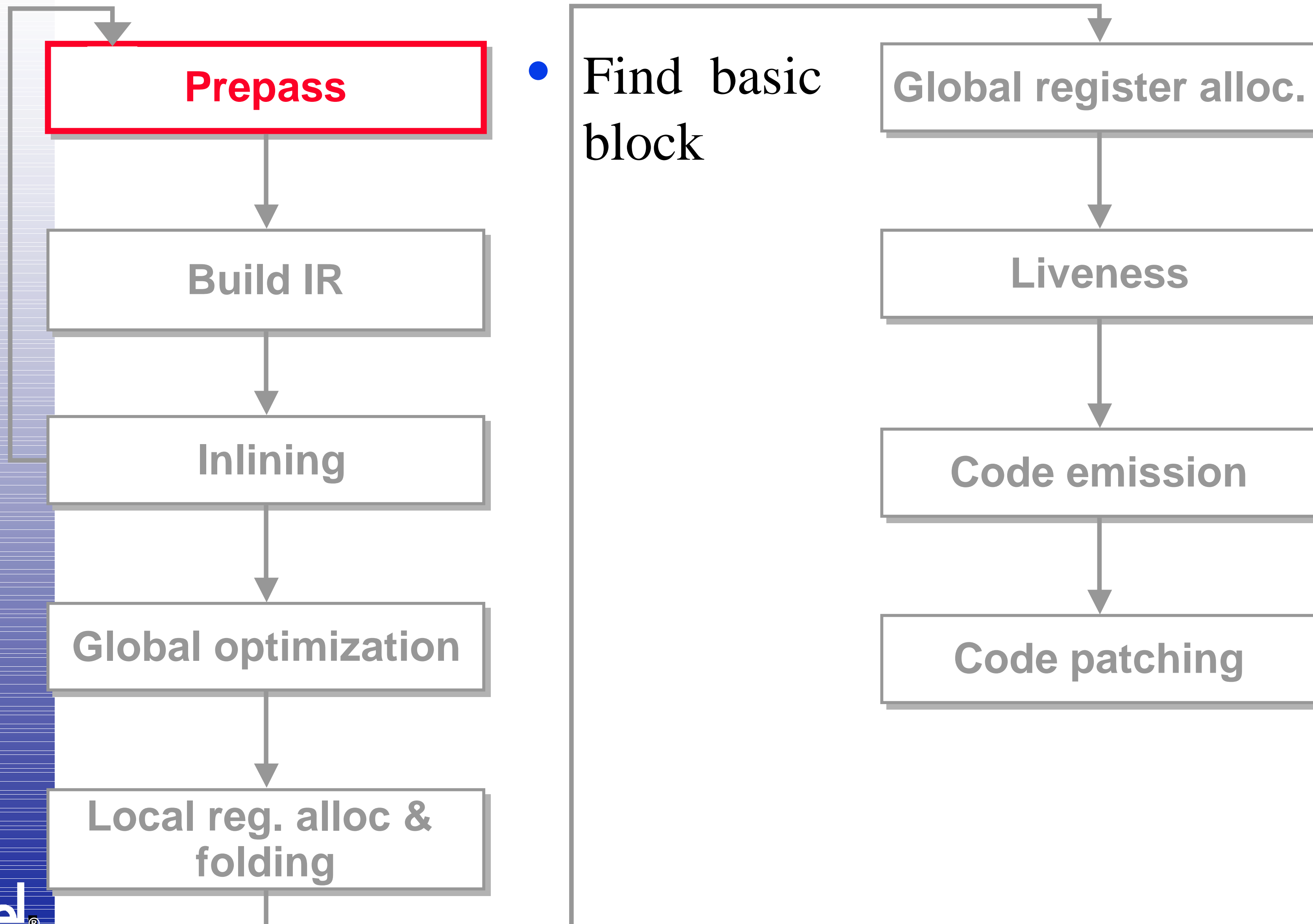
Outline

- Fast code generation
- Optimizing compiler
- Recompilation
- Usage

Structure of optimizing compiler

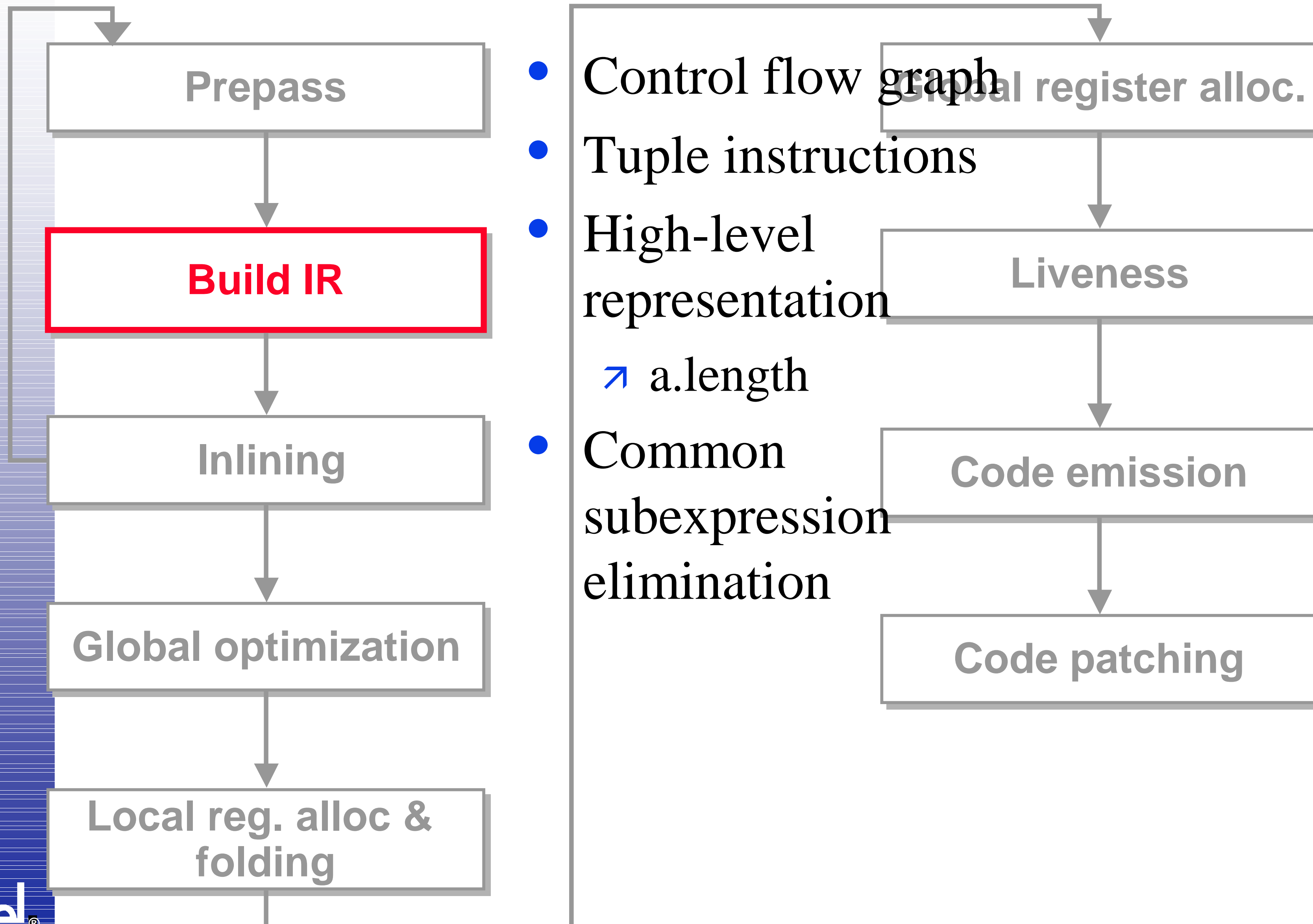


Structure of optimizing compiler



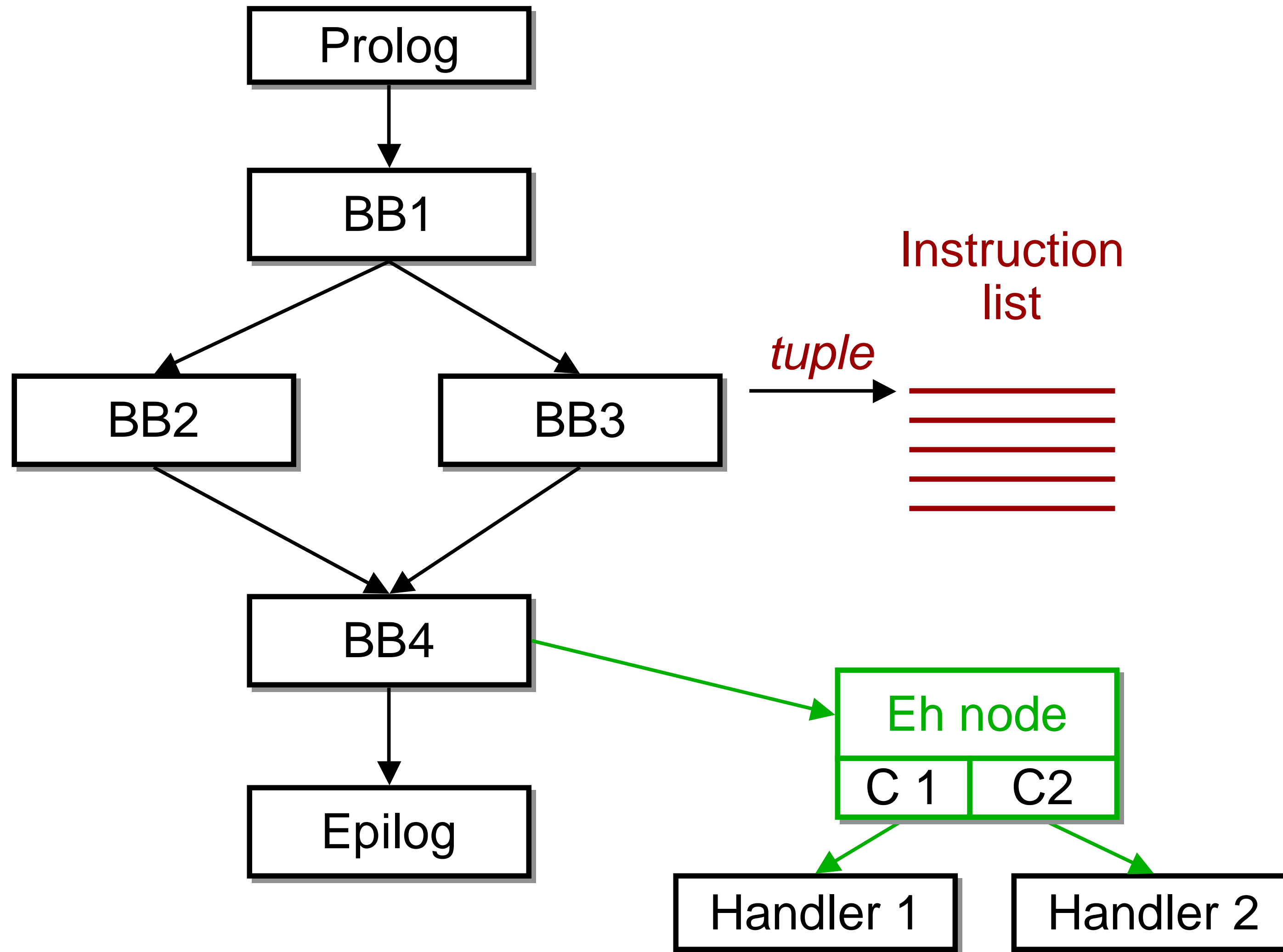
- Find basic block

Structure of optimizing compiler

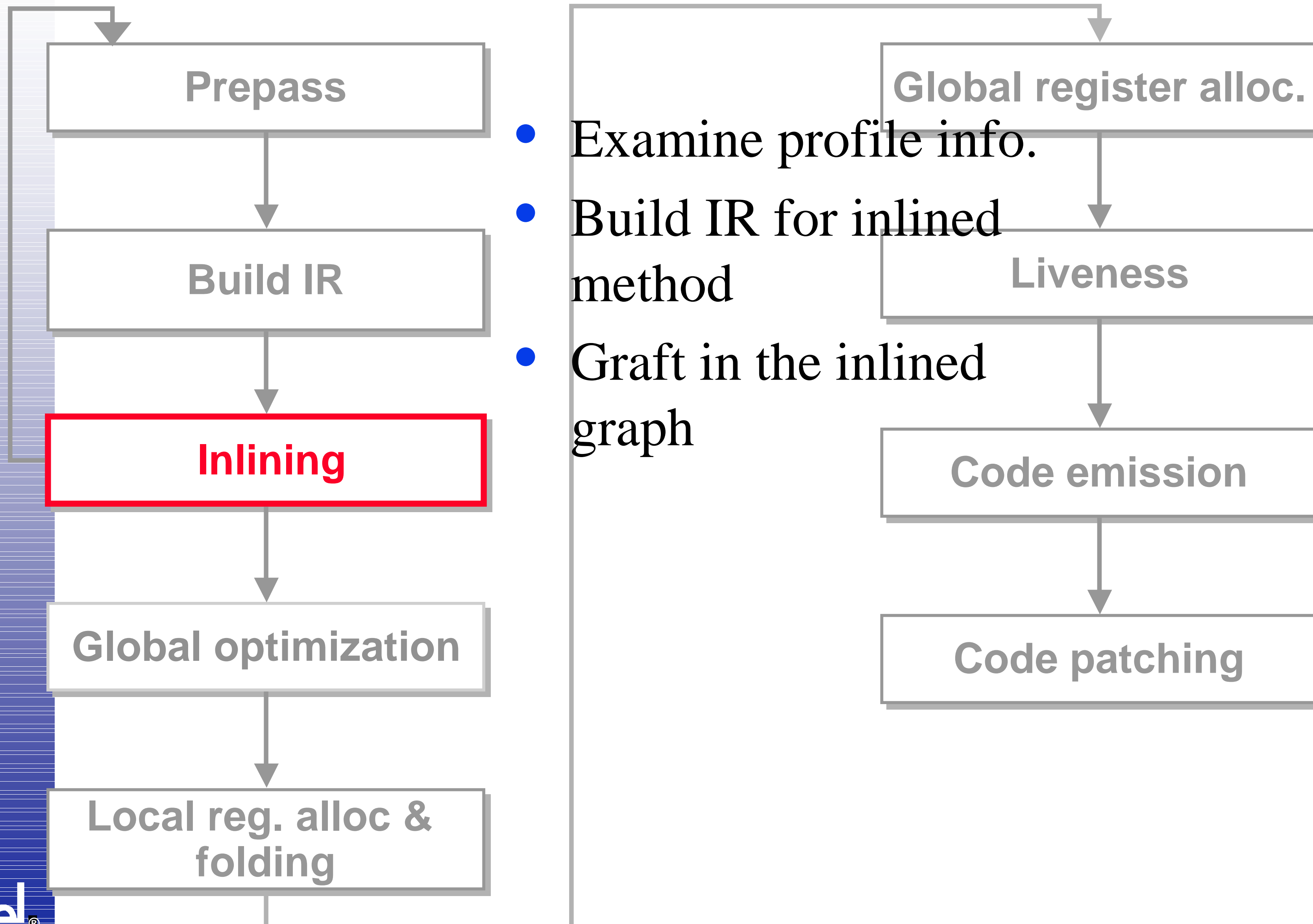


- Control flow graph
- Tuple instructions
- High-level representation
  - a.length
- Common subexpression elimination

CFG structure

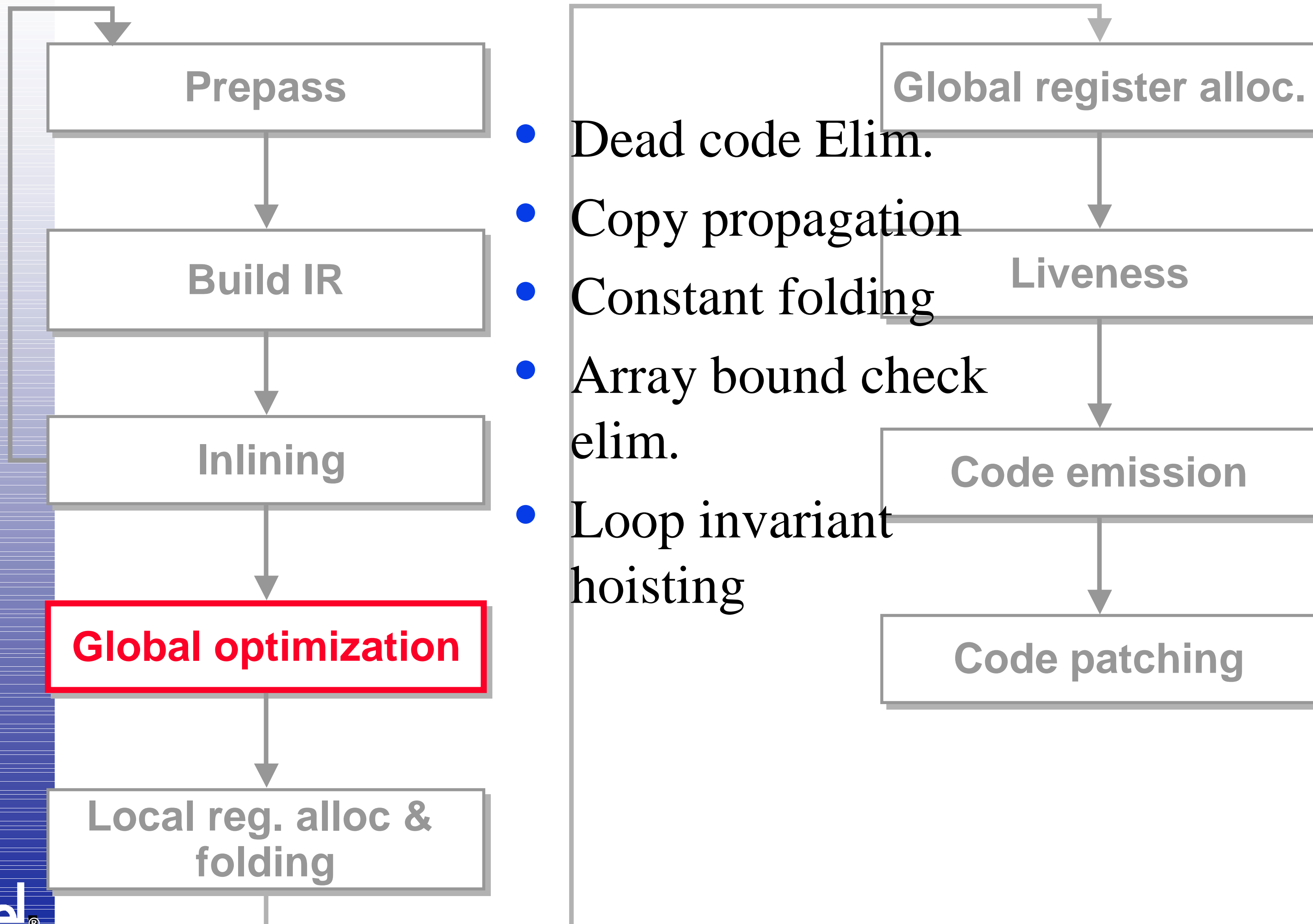


Structure of optimizing compiler



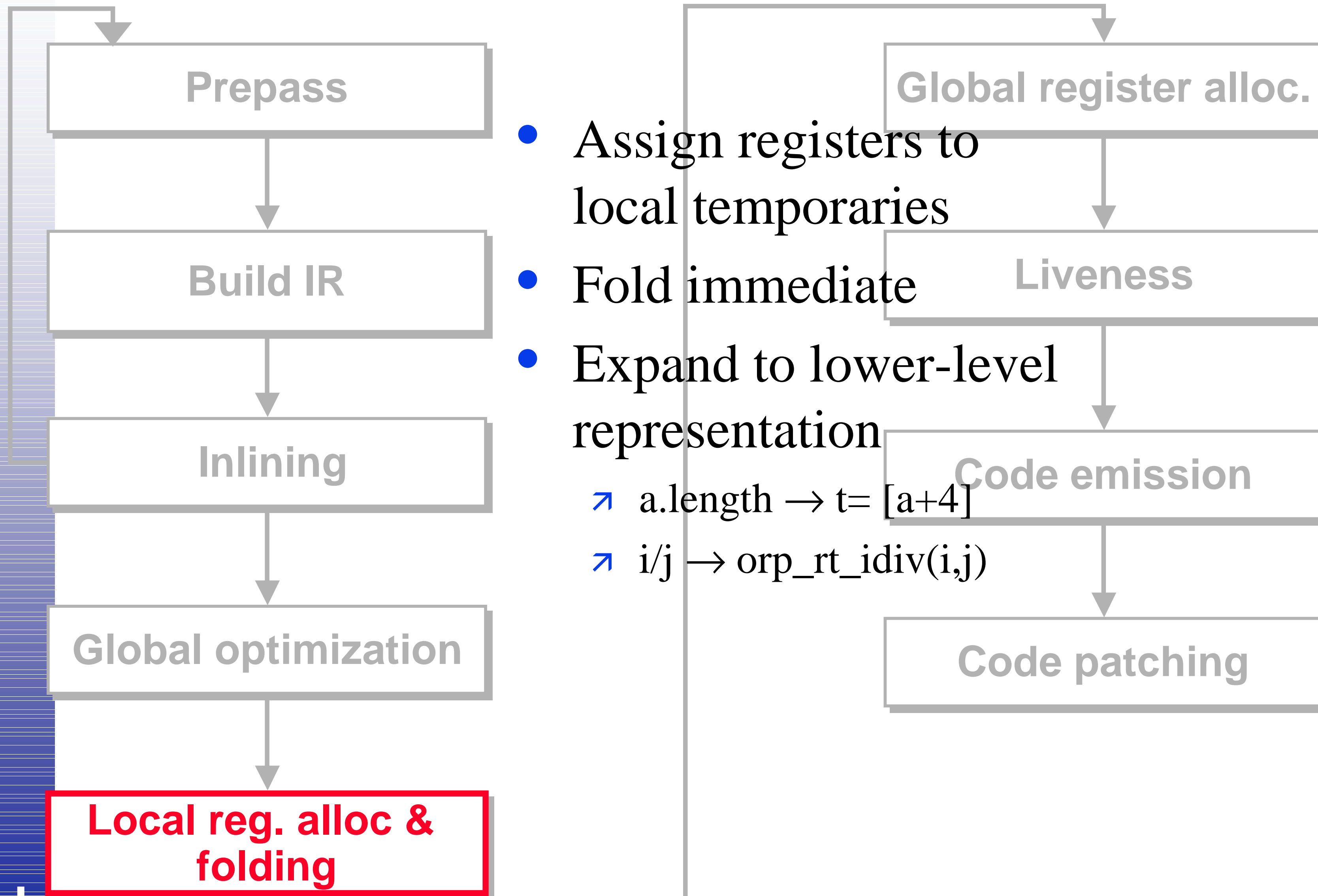
- Examine profile info.
- Build IR for inlined method
- Graft in the inlined graph

Structure of optimizing compiler

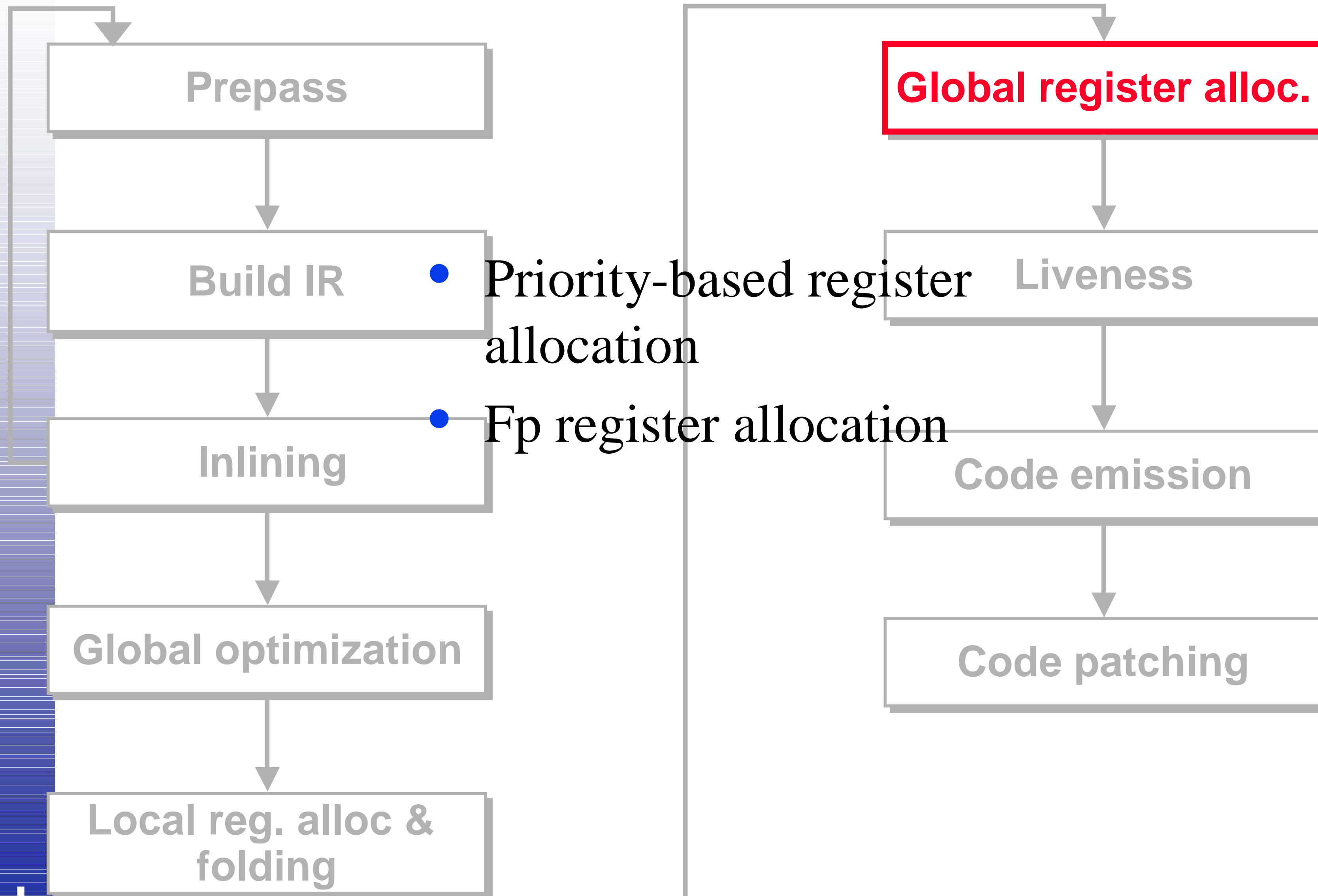


- Dead code Elim.
- Copy propagation
- Constant folding
- Array bound check elim.
- Loop invariant hoisting

Structure of optimizing compiler

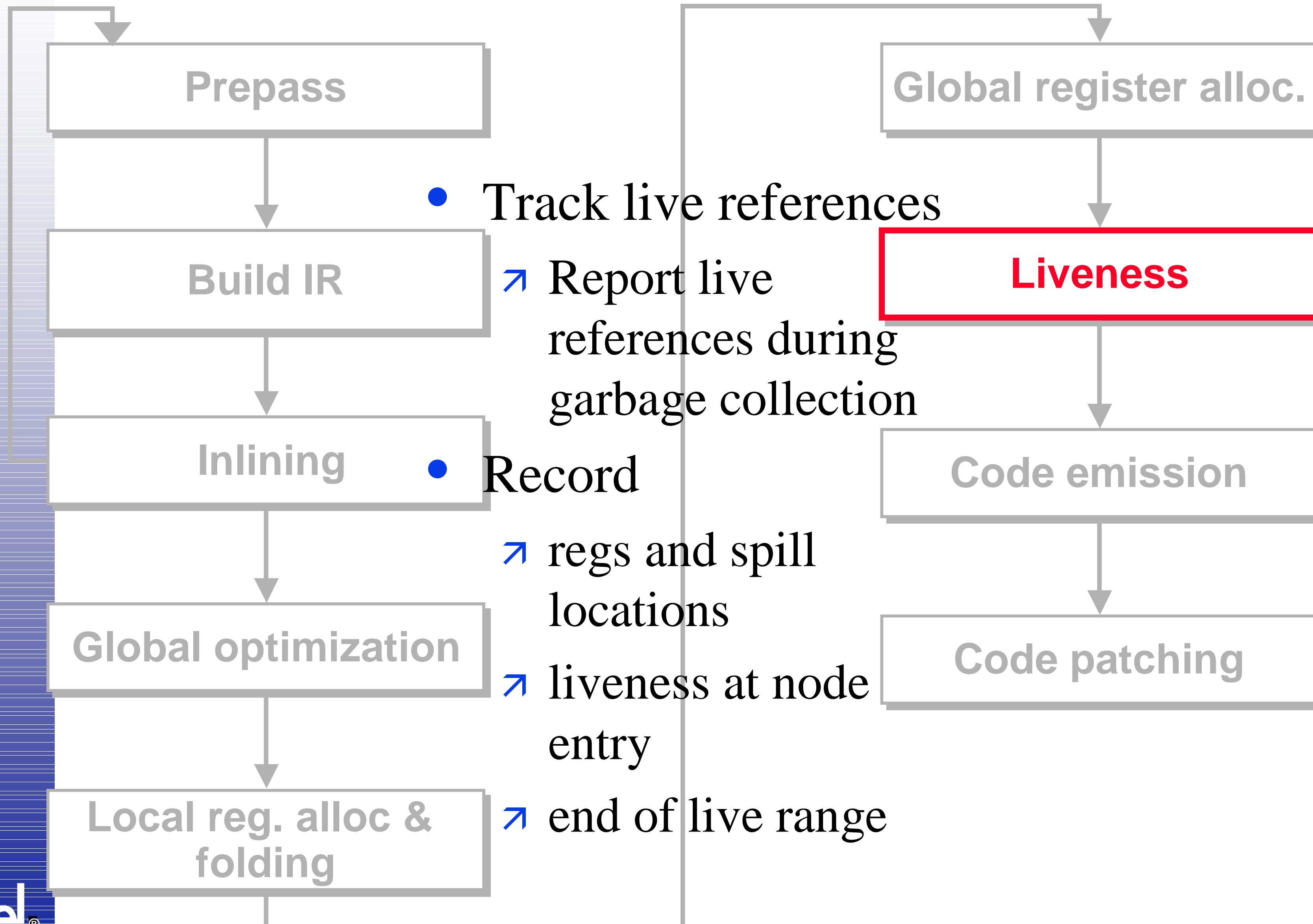


Structure of optimizing compiler



- Priority-based register allocation
- Fp register allocation

Structure of optimizing compiler



- Track live references
  - Report live references during garbage collection
- Record
  - regs and spill locations
  - liveness at node entry
  - end of live range

Outline

- Fast code generation
- Optimizing compiler
- Recompilation
- Usage

When to recompile

- Eagerly
  - Compilation time is wasted
- Lazily
  - Performance suffers
- Loop intensive
  - Counters for back edges
- Call intensive
  - Counters for method entry points

How to trigger recompilation

- Instrumenting
  - Set up thresholds for counters
  - Decrement counters
  - Trigger if a counter is zero
  - Recompile as soon as thresholds are reached
  - Execution and compilation do not overlap
  - Choosing right thresholds is hard

How to trigger recompilation

- Separate thread
  - Scan method info (counters)
  - Trigger if a method is hot
  - Execution and compilation overlap
  - Not recompile immediately
- Combine the two
  - Instrumenting + thread
  - Overhead of creating threads

Misc. uses of recompilation

- Just-In-Time GC support
  - Generate GC support when needed
  - Time and space tradeoff
- Just-In-Time debugging support
  - Stack frame access
  - Data-value access
  - Control breakpoint
  - Data breakpoint
- Recycle code space

Outline

- Fast code generation
- Optimizing compiler
- Recompilation
- Usage

Usage: Multiple JITs

- JIT *jit_compilers[]
  - jit_compilers[0]: o3_jit
  - jit_compilers[1]: o1_jit
- Compilation:
  - Start from jit_compilers[0]
  - Default: o3_jit
- **`-swapjit 0 1`**
  - Default: o1_jit
- **`-swapjit 0 1 -jitO1a instrument`**
  - Dynamic recompilation

intel®

Usage: Selective compilation

- -jitO3 METHODS=list:
  - o3_jit writes method names to file list only
  - o1_jit compiles all methods
- -jitO3 METHODS=list:5-10
  - o3_jit compiles 5-10th methods of the list
  - o1_jit compiles the rest methods
  - Binary search to find out which method has bugs
- Check mtable.cpp for more options

intel®

Usage: Dumpjit

- Compilation define _DEBUG, _DUMP_JIT
- dumpjit.txt in working directory

```
32: caload
   040C1AC3 mov [esp+20] -> ecx                          8b 4c 24 14
   040C1AC7 movzx short [ecx+8+eax*2] -> eax 0f b7 44 41 08
 33: ldc #1 int 65280
 35: if_icmpne 50
   040C1ACC cmp 0xff00 -> eax                            3d 00 ff 00 00
   040C1AD1 jne 040C1AE4                                 0f 85 0d 00 00 00
```

Usage: Dotfiles

- Compilation define: _DEBUG, DUMPJIT, PRINTABLE_O3, TRACE_O3
- -jitO3 DOTFILES=list:5,foo
  - mtable.cpp
- Produce dot files in dotfiles
  - Control flow graph
  - IR instructions
  - Different phases
- Dot.exe
  - Graphviz from AT&T
  - Convert dot to postscript

Conclusion

- Mailing list (http://groups.yahoo.com/group/orp)
- The source code itself (http://intel.com/research/mrl/orp)
- BSD like license