Sun Hotspot JVM

Part 1: The Interpreter

... on the x86 platform

All trademarks and registered trademarks are the property of their respective owners.
All Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

How it works generally

- The interpreter is generated at runtime by the InterpreterMacroAssembler (a subclass of MacroAssembler, and Assembler)
- There are two dispatch tables (with max 256 entries)
  - One is the normal mode table
  - The other is used to bring the interpreter to a safepoint (e.g. when a GarbageCollection should be made, or synchronization)

Why runtime generated ?

- The advantage of RT-generated is, you can do it optimally for this type of CPU (486, Pentium 1/2/3/4, SSE1/2 etc)
- The x86 hasn't got much registers – Sun knew which values have to stay in regs, and which ones on stack:
- The 'Top-of-Opstack' of the can be either in register RAX, or saved on the usual control stack
- Well, there is a C version around, still (cInterpreter.cpp)

Top of **operand stack**

- We have following types around:
  - atos – Object, Array
  - btos - Byte, Boolean
  - ctos - Char
  - stos - Short
  - itos - Integer
  - ltos - Long
  - ftos - Float
  - dtos - Double
  - vtos – Void (Top of opstack is not in RAX, but on stack)

How is that interpreter then generated, huh?

- Below the whole thing is a CodeBuffer / Blob, which is being filled by the Assembler class (which really just emits machine code)
- There are generator functions for each kind of Bytecode(BC)
  - hey, iconst_0 isnt that different from iconst_5, right ?
  - some of them are complex beasts, but all take just 1 arg you'll see..
  - all hidden in TemplateTable.cpp, and its platform dependent (PD) file TemplateTable_i486.cpp

some examples

```
void TemplateTable::iconst(int value) {
  transition(vtos, itos);
  if (value == 0) {
    __ xorl(rax, rax);
  } else {
    __ movl(rax, value);
  }
}
```

```
void TemplateTable::bipush() {
  transition(vtos, itos);
  __ load_signed_byte(rax, at_bcp(1));
}
```

```
#define __  masm->
#define transition(instate, outstate)
/* some assert function to check that both states are
really conforming to what that Bytecode is supposed
to do – defined for each Template*/
```

```
void TemplateTable::laload() {
  transition(itos, ltos);
  // rax,: index
  // rdx: array
  index_check(rdx, rax);
  __ movl(rbx, rax);
  // rbx,: index
  __ movl(rax,
          Address(rdx, rbx, Address::times_8,
          arrayOopDesc::base_offset_in_bytes(T_LONG) +
          0 * wordSize));
  __ movl(rdx,
          Address(rdx, rbx, Address::times_8,
          arrayOopDesc::base_offset_in_bytes(T_LONG) +
          1 * wordSize));
}
```

// For each kind of argument (int, boolean, nothing) there is a 'def' function, which casts the argument and calls:

```
void TemplateTable::def(Bytecodes::Code code, int flags, TosState in, TosState out, void (*gen)(int arg), int arg) {
  // should factor out these constants
  const int ubcp = 1 << Template::uses_bcp_bit;  const int disp = 1 << Template::does_dispatch_bit;
  const int clvm = 1 << Template::calls_vm_bit;   const int iswd = 1 << Template::wide_bit;

  bool is_wide = (flags & iswd) != 0;   // determine which table to use
  assert(in == vtos || !is_wide, "wide instructions have vtos entry point only");
  Template* t = is_wide ? template_for_wide(code) : template_for(code);
  t->initialize(flags, in, out, gen, arg); // its just saved here.. the code gen is called later..
}
```

```
void TemplateTable::initialize() {
  if (_is_initialized) return;
  // this is the big initialization table, which calls the right def, which in turn calls later the PD generator for it...
  // Java spec bytecodes              ubcp|disp|clvm|iswd    in    out   generator     argument
  def(Bytecodes::_nop                , ____|____|____|____, vtos, vtos, nop          ,  _      );
  def(Bytecodes::_aconst_null        , ____|____|____|____, vtos, atos, aconst_null  ,  _      );
  def(Bytecodes::_iconst_m1          , ____|____|____|____, vtos, itos, iconst       , -1      );
  def(Bytecodes::_iconst_0           , ____|____|____|____, vtos, itos, iconst       ,  0      );
  ...
  def(Bytecodes::_iconst_3           , ____|____|____|____, vtos, itos, iconst       ,  3      );
  ...
  def(Bytecodes::_fadd               , ____|____|____|____, ftos, ftos, fop2         , add     );
  ...
  def(Bytecodes::_ifgt               , ubcp|____|clvm|____, itos, vtos, if_0cmp      , greater );
```

How it really works...

- init_globals calls ::interpreter_init(), which calls Interpreter::initialize(), therefore creating an Interpreter object, the call gets dispatched to AbstractInterpreter::initialize() which does:
  - **TemplateTable::initialize()** // Prepare the Templates into array
  - _code = new StubQueue(new InterpreterCodeletInterface, ...); // where the machine code gets stored..
  - **InterpreterGenerator(_code)**; // its a stack obj, which creates all the code & stubs !
  - active_table = normal_table; // and sets the right dispatch table

How it really works... Pt 2

- The constructor of InterpreterGenerator, by his super constructor sets some values to NULL, and calls AbstractInterpreter-Generator::generate_all() which generates (in order):
  - unimplemented BC handler, illegal_BC_sequence (~> exit(1))
  - trace_code(TosState) // push TosState, call SharedRuntime::traceBC
  - return_entry(TosState, length) // jump back into interp. from child method
  - early_ret(TosState) // forced return by debugger/JVMTI, removes activation frame, puts assignment compatible result on stack
  - deopt_entry(TosState, length) // when a compiled method is forced to be run again in the interpreter, this is used as re-entry point into interpreter, very similar to the normal return_entry

How it really works... Pt 3

- the array of BasicTypes // T_XXX, as JVM_T_XXX in jni.h defined
- only with C-Interpreter: the tosca2stack, stack2stack, stack2native converters
- slow signature handler // sets up the arguments for a native call, by calling into the VM – used if no fast handler is around (too many arguments)
- the return_(3|5)_addrs_index // = copy of return_entry(TosState, length=3|5)
- continuation handler(TosState) // part of the deoptimization process
- safept_entry(TosState) // calls InterpreterRT::at_safept in VM, and blocks on return by calling thread->handle_special_runtime_cond()
- the basic exceptions (NullPtr, StackOverflow, ClassCast, Arithm., ArrayOutOfBounds)

How it really works... Pt 4

- method_entry points for each type of method // (zerolocals = normal, zerolocal_sync, empty, accessor, abstract, trigonometric, native, native_sync)
- finally calls:

```
void AbstractInterpreterGenerator::set_entry_points_for_all_bytes() {
 for (int i = 0; i < DispatchTable::length; i++) {
  Bytecodes::Code code = (Bytecodes::Code)i;
  if (Bytecodes::is_defined(code)) {
   set_entry_points(code);
  } else {
   set_unimplemented(i); // sets all entry points to unimplemented BC handler
...

void AbstractInterpreterGenerator::set_safepoints_for_all_bytes() // sets all the generated safepoints for the BC
```

How it really works... Pt 5

```
void AbstractInterpreterGenerator::set_entry_points(Bytecodes::Code code) {
  CodeletMark cm(_masm, Bytecodes::name(code), code);
  address bep = _illegal_bytecode_sequence;  address cep = _illegal_bytecode_sequence;
  address sep = _illegal_bytecode_sequence;  address aep = _illegal_bytecode_sequence;
  address iep = _illegal_bytecode_sequence;  address lep = _illegal_bytecode_sequence;
  address fep = _illegal_bytecode_sequence;  address dep = _illegal_bytecode_sequence;
  address vep = _unimplemented_bytecode;  address wep = _unimplemented_bytecode;
  if (Bytecodes::is_defined(code)) {
    Template* t = TemplateTable::template_for(code);
    set_short_entry_points(t, bep, cep, sep, aep, iep, lep, fep, dep, vep); // this overwrites
    the addresses where defined, and returns the others unmolested  (=illegal_BC_seq)
  }
  if (Bytecodes::wide_is_defined(code)) {
    Template* t = TemplateTable::template_for_wide(code);
    set_wide_entry_point(t, wep); // no TOS for wide!
  }
  EntryPoint entry(bep, cep, sep, aep, iep, lep, fep, dep, vep);
  Interpreter::_normal_table.set_entry(code, entry);
  Interpreter::_wentry_point[code] = wep;
}
```

```
void AbstractInterpreterGenerator::set_short_entry_points(Template* t, address& bep .......  address& vep) {
  switch (t->tos_in()) {
    case btos: vep = __ pc(); __ pop(btos); bep = __ pc();        generate_and_dispatch(t);    break;
    case ctos: vep = __ pc(); __ pop(ctos); sep = __ pc();        generate_and_dispatch(t);    break;
    ....
    case vtos: set_vtos_entry_points(t, bep, cep, sep, aep, iep, lep, fep, dep, vep);          break;
    default  : ShouldNotReachHere();                                                           break;
```

```
void AbstractInterpreterGenerator::generate_and_dispatch(Template* t, TosState tos_out) {
  int step; // the next BC instruction is offset at..
  if (!t->does_dispatch()) {
    step = t->is_wide() ? Bytecodes::wide_length_for(t->bytecode()) : Bytecodes::length_for(t->bytecode());
    if (tos_out == ilgl) tos_out = t->tos_out();
    __ dispatch_prolog(tos_out, step); // does nothing on x86
  }
  t->generate(_masm);   // generate template by calling the generator with associated arguments (cf Page 5)
  if (t->does_dispatch()) {   // advance
#ifdef ASSERT
    __ should_not_reach_here();     // make sure execution doesn't go beyond this point if code is broken
#endif // ASSERT
  } else {     // dispatch to next bytecode
    __ dispatch_epilog(tos_out, step); // = dispatch_next(state, step);
  }
```

How it really works... Final Pt 7

```
void InterpreterMacroAssembler::dispatch_base(TosState state, address* table, bool verifyoop) {
  verify_FPU(1, state);
  if (VerifyActivationFrameSize) {
    Label L;
    movl(rcx, rbp);
    subl(rcx, rsp);
    int min_frame_size = (frame::link_offset - frame::interpreter_frame_initial_sp_offset) * wordSize;
    cmpl(rcx, min_frame_size);
    jcc(Assembler::greaterEqual, L);
    stop("broken stack frame");
    bind(L);
  }
  if (verifyoop) verify_oop(rax, state);
  Address index(noreg, rbx, Address::times_4);
  ExternalAddress tbl((address)table);
  ArrayAddress dispatch(tbl, index);
  jump(dispatch);
}
```

```
void InterpreterMacroAssembler::dispatch_next(TosState state, int step) {
  // load next bytecode (load before advancing rsi to prevent
    address generation interlock (AGI))
  load_unsigned_byte(rbx, Address(rsi, step));
  increment(rsi, step);   // advance rsi
  dispatch_base(state, Interpreter::dispatch_table(state));
```

Recap

- generated interpreter, by applying code generators with their respective arguments
- the code generators use the MASM to write code into the code buffer, CodeletMarks denote the start / end for one BC
- Dispatch table is #BC x #TosStates, and an illegal sequence is caught by its handler, as well as unimplemented ones – all others are 'normally' dispatched
- Calling a child method involves setting up the return address (cf prepare_invoke), and jumping to the child uses InterpreterMacroAssembler::jump_from

Ok, what about registers within interpreter ?

- **Register setup when dispatching UNLESS invokeXXX**
- RAX: Top-of-stack element (if TosState != vtos)
- RBX: The upcoming BC value (yes, those relevant 8bits)
- RCX: scratch ?
- RDX: Top-of-stack elem, if TosState = ltos
- RSI: Ptr to upcoming BC in mem (BCP)
- RDI: Ptr to locals
- RBP: frame pointer
- RSP: Topmost stack element (=lowest addr) in memory (only =TOS in vtos mode), implicitly changed by all the push/pop ops.

Ok, what about registers within interpreter ?

- **Register setup while dispatching invokeXXX** (cf TemplateTable::invokeXXX, TemplateTable::prepare_invoke)
- RAX: TosState is vtos!
- RBX: -> is being resolved by prepare_invoke, load_invoke_cp_cache_entry etc..
- RCX: -> is being resolved to the receiver, by loading top of stack (not for static)
- RDX: TosState is vtos!
- RSI: Ptr to upcoming BC in mem (BCP)
- RDI: Ptr to locals
- RBP: frame pointer
- RSP: Topmost stack element (=lowest addr) in memory

Ok, what about registers within interpreter ?

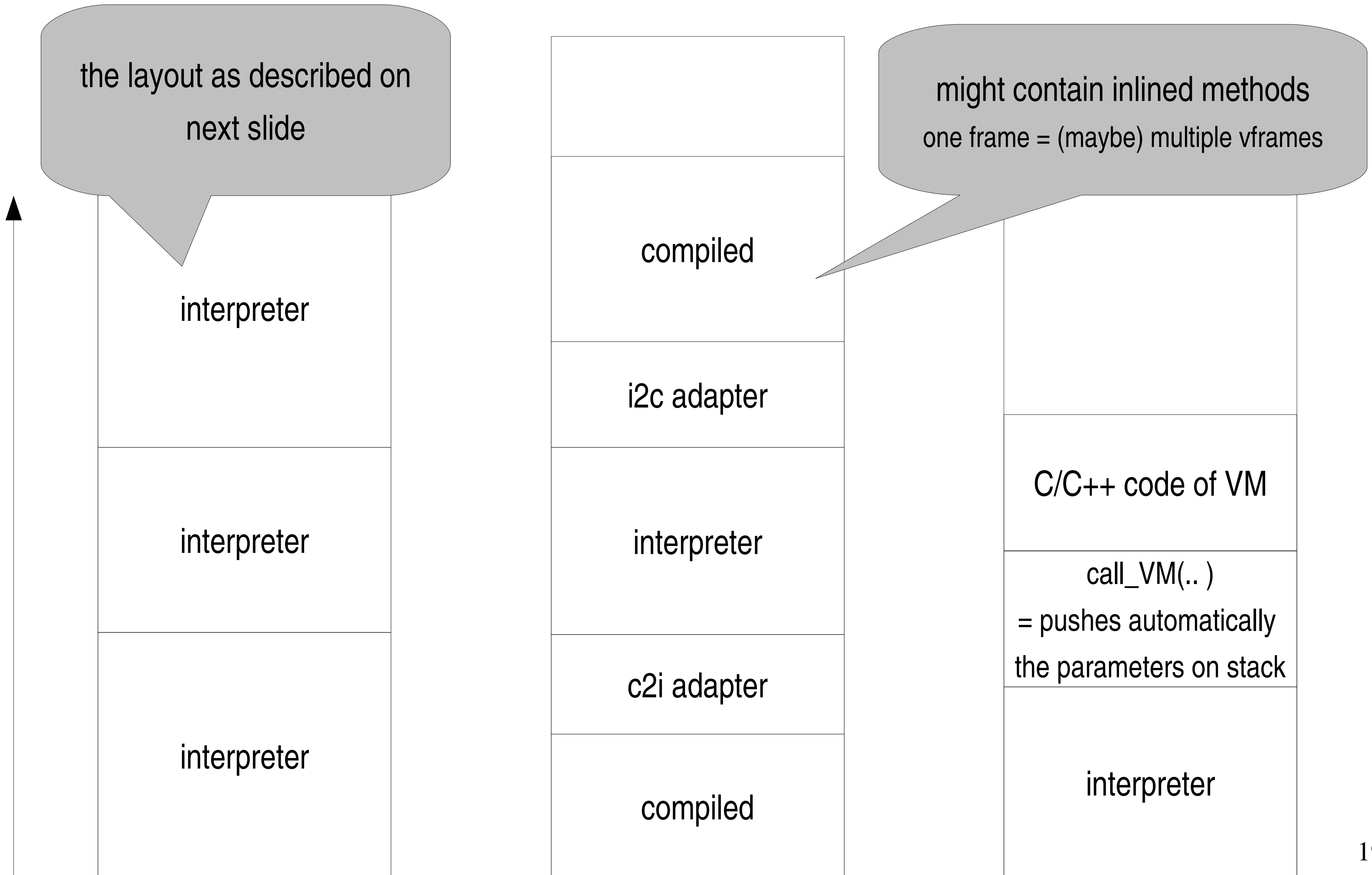
- **Register setup jumping to the new method** (cf TemplateTable::prepare_invoke)

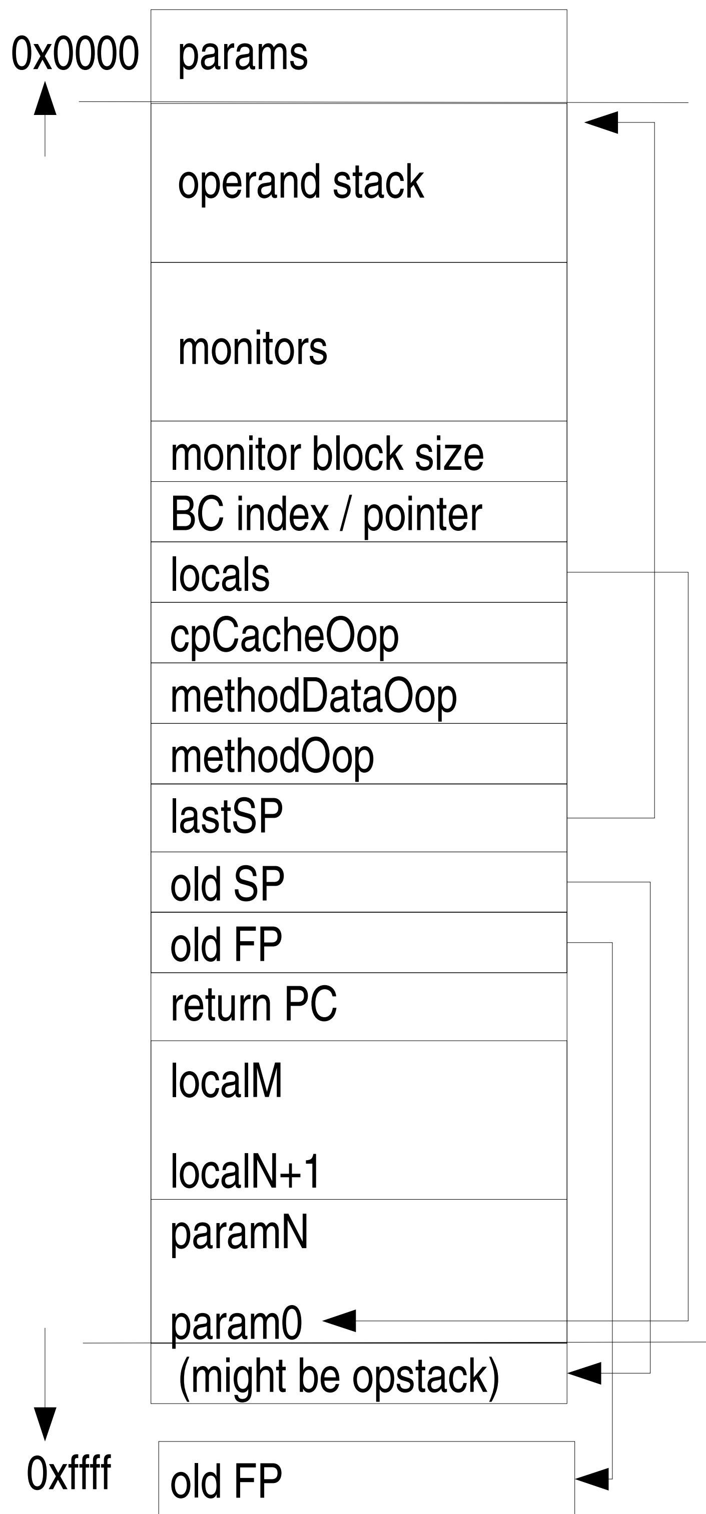
- RAX: scratch
- RBX: methodOop
- RCX: receiver (unless static), becoming scratch
- RDX: scratch

~= C calling convention (cdecl) ! RAX,RCX,RDX can be used, RBX must be callee saved, though we are not using it again

- RSI: Ptr to upcoming BC in mem (BCP) – is restored from frame on return by interp
- RDI: Ptr to locals, getting scratch – is restored from frame on return by interp
- RBP: frame pointer
- RSP: Topmost stack element (=lowest addr) in memory

some stack frames...





- operand stack: where the evaluations are done. When interpreter is running, top of stack element is typically in RAX/RDX (unless TosState = vtos). There is also a 'tagged' interpreter mode, where each element on the stack is accompanied with its type information
- monitor objects, and the
- size of the block used by them
- Bytecode index (0-length of method) / BC pointer (real addr of BC in mem)
- pointer to the 'locals' the first N are the parameters, plus extra locals
- the constantPool Cache entry for that method („runtime symbol table")
- handle to the profile of that method (for jumps, loops, etc...)
- handle to the method descriptor (BC, compiled versions, inv. counter)
- lastSP is NULL as long as the interpreter is running, when calling child method this represents top address of the opstack. The parameters are dropped from stack at return by restoring this value into ESP again
- old stack pointer / frame pointer for stack walking purposes (GC!)
- the return pointer is either the return_entry for the interpreter (for the TosState that we are leaving as result in RAX) or a stub code (c2i, deopt..)
- the locals and params

(see frame_i486.hpp)