

SABLEJIT: A Retargetable Just-In-Time Compiler for a Portable Virtual Machine

David Bélanger

dbelan2@cs.mcgill.ca

Sable Research Group

McGill University

Montreal, QC

January 28, 2004

Overview

- Introduction and Motivation
- SableVM Overview
- SableJIT Overview
- Method Invocation
- Direct and Inlined Mode
- Exception Handling
- Memory Management
- Self and Partial Compilation
- Results
- Conclusion

Introduction and Motivation

- SableVM is designed to be an efficient portable bytecode interpreter.
- Code interpretation has a significant overhead and just-in-time (JIT) compilers are designed to overcome this problem.
- Most JITs have focused on efficiency, making code very machine specific. Retargeting to a new platform involves a lot of work.
- A compilation unit is usually a method.
- `SABLEJIT` is designed to be relatively easy to port to new platforms while remaining reasonably efficient.
- Furthermore, a serious attempt is made to allow flexibility concerning the compilation unit.

SableVM - Overview

- Portable bytecode interpreter written by Etienne Gagnon.
- 3 interpreters “flavours”:
 - Switch interpreter.
 - Direct-threaded interpreter.
 - Inlined-threaded interpreter.
- Signal-based exception detection.
- Copying garbage collector.

SableJIT - Overview

- Mixed-mode interpreter / jit
- Works with all 3 interpreters
- Compiler component mostly written in Java
- Simple baseline compiler
- No optimizer
- Self-compiles
- Code relocatable

Supported Platforms

- PowerPC: Linux and Darwin / Mac OS X.
- Intel: Linux and FreeBSD.
- Sparc: Solaris (in progress).
- Relatively easy to add support for new RISC architectures.

Compiler Overview

- One-pass code generator (1 pass + patching branches)
- No IR data structure built, code generated directly.
- Architecture dependent aspects simply abstracted through inheritance.
- Both SableVM and SableJIT shares the same Java stack.
 - Locals are read/written from/to the stack.
 - Return value is written directly to the stack.
- Simplifies handling of exceptions and garbage collection significantly.
- SableJIT is in fact mimicking SableVM.

No Optimizer But...

- Uses some information from SableVM:
 - Method synchronization
 - Unrolling loop setting arguments to `null`
 - Stack offsets

Method Invokes in Interpreter

- SableVM has bytecodes:
`INVOKE{VIRTUAL, SPECIAL, INTERFACE, STATIC}`
- To reduce overhead in interpreter, SableJIT adds:
 - `INVOKE*_JITCOMPILE_BOOTSTRAP`: check if compiler initialized, check if should compile.
 - `INVOKE*_JITCOMPILE`: check if should compile
 - `INVOKE*_CALLCODEa`: no check
- Overhead when interpreting only: selection of `INVOKE*` when preparing interpreter code.

^aOnly for special and static.

Method Invokes in Compiled Code

- Compiled code always assuming methods are compiled.
- Executes `method->compiled_code` without checking if `method` is compiled.
- `method->compiled_code` is pre-set to a special function in SableVM.
 - Undo compiler assumptions
 - Setup stack and invoke interpreter
- Native methods: Not direct call but bypass interpreter.
- No overhead if compiled code invokes compiled method.
- Invoking a non-compiled method is expensive.

Input Code

- Compiler uses switch interpreter code as input:
 - Extended bytecode set.
 - Array of words, not bytes.
 - Some bytecodes split: GETFIELD_INT, GETFIELD_BOOLEAN, ...
 - Preparation sequences at the end.

Direct / JIT Mode

- Direct-threaded: Implementation addresses in code array.
- Convert direct to switch code.
 - Map addresses to integer bytecode instruction.
 - Adjust addresses: branches, jump tables, ...
 - Operands copied verbatim.
- Conversion done once.
- Equivalent switch code kept and reused if needed (recompilation).

Inlined / JIT Mode

- Inlined-threaded: Addresses to bytecode sequence implementation.
- Convert inlined to switch code.
- Difficult to implement.
- Conversion more expensive.
- Array sizes mismatch, organization of operands etc.
- Requires mapping array for exceptions (*switch pc* \rightarrow *inlined pc*).
- Conversion done once.
- Kept if needed later (recompilation)

Recompilation

- Slow and fast version of bytecodes.
- SableVM has preparation sequences at end and patches interpreter code.
- SableJIT does not patch native code.
- Instead, patches interpreter code (for direct or inlined, the intermediate array)
- Profiles slow code and recompiles method.
- Advantages:
 - Simpler
 - Does not require architecture specific code.
 - Recompiled code requires less space.
- Framework in place for later with optimizer.

VCODE-like Backend

- RISC-like instruction set (ex: `add rd, rs, rt`).
- Each instruction correspond to a method (ex: `addI(rd, rs, rt)`).
- Some instructions implemented in terms of others (ex: `mulI`).
 - Allows a quick port.
 - Override for efficiency.
- System independent branch resolution and patching.
- System independent tableswitch, tablelookup, ...
- x86
 - CISC architecture (ex: `add rd, rs`)
 - Only 8 “GPR” registers. Stack used as registers.

Porting Approach

- Port the `VCODE` instruction set to your platform.
- Does not require knowledge about the layout of objects, arrays, fields etc.
- Knowledge of an architecture is sufficient for an initial port.
- Changes in SableVM object layout or bytecode instructions require minimal changes in SableJIT.

Compilation - Robustness

- If an exception is thrown during compilation
 - SableJIT performs any required cleanup.
 - Method is marked as uncompilable.
 - If recompiling, will use the last successful compilation.
 - Otherwise, interpret.
- Very useful for incremental development and testing.
 - Parts of implementation can be delayed
(`NotImplementedException`)
 - Running test suites

Exception Handling - Overview

- Exceptions thrown in compiled code are handled by the interpreter.
- SableJIT and SableVM uses same stack.
- Leave stack untouched and return control to interpreter:
 - Simple return
 - With signals

Exceptions - Simple Return

- If an exception is thrown by the compiled code, the compiled code:
 - Set the exception object in `env->throwable`
 - Saves the bytecode pc in the stack frame.
 - Returns `-1`.
- Control will get back to interpreter eventually.
- If the compiled code returns `-1`, the interpreter jumps to the exception handler.

Exceptions - In SableVM

```
switch (bytecode) {  
  
    case INVOKE* :  
        ...  
        stack_inc = compile_code(...)  
        if (stack_inc == -1) {  
            goto exception_handler;  
        }  
        ...  
}  
  
exception_handler:  
    /* Code to handle exception */
```

Signal-based Exceptions - Overview

- SableVM can have signal-based exceptions.
- Most processors generate an exception (or trap) on invalid operations (ex: dereferencing null pointer, division by zero)
- Idea: Do not check for exceptional case, simply go ahead.
- OS will send SIGSEGV, SIGFPE, ... to SableVM.

Signal-based Exceptions - SableVM

- In the interpreter:

```
/* save pc in case exception is raised */  
env->stack.current_frame->pc = pc;  
/* get integer field */  
stack[stack_size - 1].jint =  
    *((jint *) (((char *) instance) + offset))
```

- Note: Required to save bytecode pc. Interpreter implementation is shared between methods.

Signal-based Exceptions - SableJIT

- Note: Memory address of the fault uniquely identify the location.
- Not required to save bytecode pc.
- Compute and store mapping (native address^a, bytecode pc) at compilation time.
- If exception occurs, compute bytecode pc from native pc using mapping.
- Storage required: 2 words per null check.
- Note: Requires more space if signals not used, extra instructions.

^aactually, an offset from the start of compiled code

ArrayBoundsException - powerpc

- Need a single unsigned compare:

```
if (index >=U size) {  
    throw exception  
}
```

- powerpc has a trap word instruction: `tw cond, r3, r4`
- Compare 2 registers and generates a hardware trap if condition is true.
- Process will receive a SIGTRAP.

Memory Management

- Memory manager under development.
- Delegates most allocation to class loader.
- Centralize place.
- Data structures whenever possible are allocated on Java heap.

Garbage Collection

- Solution: Since we work directly on the Java stack, no need to maintain extra roots.
- Needs to store the state at some specific points where GC could be triggered.
 - Bytecode pc
 - Stack size
- These points same as interpreter.
- Compilation may trigger garbage collection.
- General idea is simple but difficult to debug.

Garbage Code Collection

- Recompilation implies several native code for a method.
- Need to be freed.
- Need to be careful, can still be in used. Other thread or current thread (recursion)
- Option 1: add a counter `method->use_count`. Increment counter on method entry, decrement on exit. Not explored, expensive.
- Option 2: from time to time figure out code no longer used and free it.

Garbage Code Collection

- Maintain list of potentially unused code.
- At GC, when threads are stopped, record *native* pc of each thread.
- Walk *native* stack, collecting return addresses of functions.
- For each code sequence in the potentially unused code, check if some thread is in that code. If none, can be freed.
- Note: Threshold may be set.

Self-Compilation

- SableJIT can fully compile itself.
- SableJIT can start compiling as soon as SableVM has bootstrapped.
- Need to avoid compilation loops
- Need to avoid class loading loops
- Need to turn off compilation at some points in VM. Mostly related to class loading.
- Code needs to be re-entrant (several compilation in progress even if single-threaded)
- Could use knowledge about itself: Inlining virtual invokes

Partial Compilation

- Example: Compiling hot loops

```
public int __compile_me_partial() {  
    int sum = 0;  
    for (int i = 0; i < 100000; i++) {  
        sum += i;  
    }  
    return sum;  
}
```

- Profile back edges
- Send code fragment to compiler
- Relatively easy to implement
- Trace-based JIT: Segments spanning several methods.

Compilation Entry Point

- Conceptually, can compile code at any GC check point^a.
- Conceptually, can enter compiled code at any bytecode.
- SableJIT compiles/recompiles full methods on back edges and jump in the middle of it. Experimental.

^aRemember SableJIT is written in Java, it can trigger gc...

Results - Linux/ppc

Benchmark	switch	direct	inlined	switch-jit	direct-jit	inlined-jit
compress	312.98	234.95	160.95	134.91	131.74	133.26
jess	76.22	63.09	56.60	57.75	57.38	57.62
raytrace	90.07	74.30	67.78	85.48	84.88	85.72
db	157.84	133.15	108.75	104.49	106.53	108.48
javac	113.92	95.74	87.35	112.28	108.51	failed
mpegaudio	279.46	206.40	159.41	114.36	115.49	115.15
mtrt	93.85	77.26	71.42	94.20	92.13	95.60
jack	47.83	41.77	39.98	45.39	failed	failed
sablecc	96.28	80.87	71.42	87.13	81.45	79.18

Conclusion and Further Work

- We have presented an overview of SableJIT.
- We have seen SableJIT/SableVM interfaces with SableVM.
- SableJIT handles exceptions efficiently.
- SableJIT collects no longer used code in the presence of recompilation.
- Further Work
 - Cleanup code and improve implementation.
 - Add simple profiler
 - Add an optimizer
 - Port to new platforms.