Cog Blog

Speeding Up Terf, Squeak, Pharo and Croquet with a fast open-source Smalltalk VM

{ 2008 06 06 }

Cog

So Hi!

I'm delighted to say that Qwaq has taken me on to write a fast Croquet VM and that the VM is to be released under the Qwaq open source licence (an MIT license). I'm going to blog about the VM here as I implement it. The blog is a chance for me to record design decisions as I go along, receive better ideas from you, dear reader, and to hand out the whitewash.

This VM will have a dynamic translator, or JIT for short. It will dynamically compile Smalltalk bytecodes to machine code transparently to the programmer, and execute this machine code instead of interpreting bytecode. Initially I'm aiming at performance equivalent to the VisualWorks VM, hps. This VM should execute pure Smalltalk code some 10 to 20 times faster than the current Squeak VM. Subsequently I hope to do adaptive optimization and exceed VisualWorks' performance significantly. I expect to be able to release the first fast version within a year.

The VM is called Cog after the Honda advert. Fast VMs tend to have something of the Heath Robinson or Rube Goldberg about them. But one of my rules in Cog (rules are meant to be broken) is to do the simplest thing that could possibly work (TSTTCPW). My principles for Cog, in the sense of fundamental propositions governing my behaviour, are that

– any changes in the image to accomodate the fast VM will not slow down the standard Croquet VM (the interpreter)

– any changes in the image to accomodate the fast VM will not increase overall footprint

– the fast VM will be compatible with the Hydra VM

– there will be some form of source code compatibility for Slang plugins so they can function both in the interpreter and in the fast VM

I want the Cog to be used by as many people as possible, but I realise there are those interested in small machines or in VM experimentation for whom the interpreter will still be a better choice. I hope the principles above will avoid schism and mean the Squeak community can still use a common image format and preserve image portability.

That said I do intend to change the image format somewhat. I've already implemented closures. These, when implemented appropriately, have a significant impact on the performance of a JIT VM. This has meant a slight change in the bytecode set; gone are the 6 experimental bytecodes 139 through 143. I'll blog on closures in a subsequent post quite soon.

Hydra is an interesting approach to concurrency, and something Qwaq is interested in. BTW, I think Hydra would better be called String, strings being composed of threads or fibers. String, like a cog, is typically a small part of a larger whole.

For the detail-oriented let me dive down a level and say a little more about the project, especially preparatory changes that pave the way for a fast VM. I apologize for the density. I'm interested in knowing what level of detail I should go into. Obviously people who want to collaborate on Cog will want a high level of detail. But if the architecture of a fast VM is of more general interest then I need plenty of feedback to help me explain things at the right level of detail. Please feel free to comment on this!

OK then, some details. Both TSTTCPW and the Principles imply keeping the existing CompiledMethod format. In Squeak, as in Blue Book Smalltalk-80 CompiledMethod is an anomalous hybrid, its first part being object references used mainly for literals, and its second part being raw bytes used mainly for bytecodes. This is a good choice for an interpreter since both literals and bytecodes can be fetched from the same object, but it causes minor complications in the garbage collector and gets in the way of adding instance variables to CompiledMethod and hence subclassing. I'll blog on why I'm keeping the format and how to get round its limitations soon.

The existing bytecode set is all very well but inflexible. It has pressing limits on the size of blocks (1023 bytes of bytecode) and the number of literals (255). Being designed for a 16-bit system it misses some compact encoding opportunities, such as a bytecode that pushes a given SmallInteger other than -1, 0, 1 & 2. Pushing 3 requires a minimum of 1 byte for the push literal bytecode plus 4 bytes for the literal slot holding the object 3. A two byte bytecode could push -128 to 127 but take only 2 bytes and be faster. Luckily migrating to a new bytecode encoding is quite easy. Most opcodes – a name I'll use for an abstract operation such as pushInstVar: – occur in multiple bytecodes – a name I'll use for a specific encoding such as 0-15, 0000iiii, Push Receiver Variable #iiii. For all short-form encodings there is an equivalent long-form, and one can modify the compiler to generate only the long forms, allowing the ranges allocated to short-form encodings to be reused. I've already recompiled a 3.8 image to long-form.

One of the most important optimizations in a fast VM is that of context-to-stack mapping. Conceptually every method or block activation in Smalltalk is represented by a context object, each of which has its own small stack holding its temporaries and intermediate results. Each context points to its caller via a sender slot. This has lots of advantages, such as ease of implementing the debugger, implementing an exception system, being able to implement "exotic" control structures such as coroutines and backtracking, and persisting processes, all with no VM support beyond unwind-protect. The downsides are that a naive implementation incurs considerable overhead. Each send involves allocating a new context and the moving of receiver and arguments from the caller to the callee context. Each return involves (eventually) reclaiming the returned-from context, *unless* of course something has kept a reference to it. Compared to stack organization in conventional languages, contexts are sloooow.

The idea in context-to-stack-mapping (you've guessed it, details in a subsequent post) is to house method and block activations on a conventional stack, hidden in the VM and invisible to the Smalltalk programmer. Sends look more like conventional calls, passing arguments in the stack in the conventional way, the callee directly accessing the arguments pushed by the caller on to the stack. The VM then creates contexts only when needed. To the Smalltalk programmer contexts still exist, and have the same useful semantics as ever, but the overheads for the common case (send and return) are much reduced. While context-to-stack mapping is essential to a fast VM that still provides contexts, it is also useful in an interpreter. Andreas Raab suggested I implement context-to-stack mapping in the interpreter as a milestone on the way to the JIT VM. I think this is a great idea. It should provide a decent speed-up to the Squeak interpreter and help motivate things like the closure implementation.

I think that's enough for a first post. Let me wrap up with a road map of the project "going forward". None of this is set in stone, but it's a plan.

Main line:

– restructure the bytecode compiler to decouple parse nodes from specific bytecode encodings (complete)

– replace non-reentrant BlueBook blocks with closures using the restructured compiler (~ 75% complete, currently working on temp names in the debugger)

– implement context-to-stack mapping in the interpreter (to be released Septemberish)

– implement a JIT to replace the interpreter on x86 (to be released Aprilish)

In parallel:

– design a bytecode set that uses prefix bytecodes to eliminate limits and provides encodings that provide benefits for 32 and 64-bit images

– provide the ability to subclass and add/remove instance variables to CompiledMethod and subclasses while retaining the compact interpreter-friendly hybrid format

– target the JIT to other ISAs such as ARM, THUMB, PowerPC and x86-64

– general VM improvements such as

– add per-object immutability support, work I've already done for Squeak at Cadence, and previously for the VisualWorks VM

– move garbage collection out of allocation and into the primitive failure code for new, new: basicNew basicNew: et al so that the VM doesn't have to deal with moving pointers all over the place. – add primitive error codes so the reason for a primitive's failure is communicated

Posted by admin on Friday, June 6th, 2008, at 1:20 pm, and filed under Cog.
Follow any responses to this entry with the RSS 2.0 feed.

You can post a comment.

{ 37 }
Comments

1. **Isaac Gouy** | 06-Jun-08 at 4:16 pm | Permalink

   > adaptive optimization

   iirc you had something to say about that in a presentation at Stanford?

   (And maybe move the recaptcha leftwards, I didn't notice it the first time.)

2. **Bob Westergaard** | 06-Jun-08 at 5:12 pm | Permalink

   Wow! Welcome to the blogging world. I'm looking forward to reading more about your cog.

3. **Damien Pollet** | 06-Jun-08 at 6:10 pm | Permalink

   Hi Eliot,

   are you using Squeak's closure compiler? http://www.squeaksource.com/NewCompiler.html

   ————

   Hi Damien,

   no I'm not. It was easy to extend the existing compiler as I had very specific ideas about how to compile closures and how I wanted to deal with migrating away from the standard bytecode set. It would have been extra effort for me to learn a new compiler and extend that. I was able to get closures going in the existing compiler in about a week, because I know it well. If the new compiler is a big improvement over the Smalltalk-80 one I know and love then it should be easy to change the back-end to target my closure bytecodes. But please understand my use of the old compiler is due to ignorance of the new compiler not any deep seated prejudice.

   Eliot

4. **Mayson Lancaster** | 06-Jun-08 at 7:35 pm | Permalink

   I hope that ByteSurgeon-like bytecode hacking will still be possible: interactive instrumentation of live code, and subsequent replacement, is a *very* useful technique for always-up systems.

   ————

   Hi Mayson,

   I can't see any reason why not. ByteSurgeon presumably will need some small porting effort to become aware of the new closure semantics and *bytecodes. But in my architecture bytecoded metods play a central role. I'l explain in a subsequent post but I intend that the adaptive optimization incarnation of Cog will be implemented entirely in btecode. The adaptive optimizer will be written in Smalltalk and run at the image level, taking bytecoded methods is its source, decorating them with type information mined from the JIT's inline caches and generating as its output new bytecoded methods which include additional "go-faster" bytecodes that can be compiled to simpler, faster bytecode. So ByteSurgeon-style hacking is in fact central to my longer-term architecture.*

Further, one thing I like about VisualWorks' class builder is that it uses ByteSurgeon style to deal with class shape change. When in VisualWorks a class adds or looses instance variables the system disassembles and reassembles bytecoded methods to change inst var offsets, avoiding source altogether. Someone should implement this for standard Squeak e.g. using ByteSurgeon.

Eliot

5. **Carl Gundel** | 06-Jun-08 at 9:25 pm | Permalink

   Is this an opportunity to realize some Aosta ideas?

   ____

   *Yes!! at least I hope so. But starting work on that is at least 9 months away. First the new baby then we start to teach it to run **fast*** 🙂

6. **Carlos Crosetti** | 07-Jun-08 at 6:22 am | Permalink

   Great news, good luck with your nice endeavour. Count on me when alpha code is ready to test!

7. **Les Howell** | 07-Jun-08 at 9:18 am | Permalink

   I am new to the low level of virtual machines (although I did implement a simulation of a 6809 on an Z80 a long time ago), but my coding has always been direct to the target architecture, and not at this level.

   When you move from a VM byte coded architecture to a target machine architecture (which I think is what you are describing), you tend to lose OS independence. Thus Croquet which now runs pretty well on Linux, Windows and Unix, may lose some of that capability or will require different VM's with the attendent support issues unless a lot of effort is expended to interface via a restricted library (currently OpenAL, which seems to be undergoing some restructuring, so I can no longer run Cobalt on my local machine for now).

Please do not structure your VM as Windows only, because that will further restrict open development.

Regards,
Les H

———

Hi Les,

I'm not proposing to move away from bytecodes. The Cog JIT will compile bytecoded methods to native code, to a substantial degree invisibly from the programmer (there are small pieces of evidence of the JIT that surface, but they're small – e.g. play with VisualWorks and see what evidence of the JIT there is in InstructionStream MethodContext, BlockContext et al). I'm going to maintain binary compatibility with the interpreter. So I hope your concerns won't become issues.

BTW, I think you're talking more about the Foreign Function Interface (FFI), the means by which the system is interfaced to the outside world, than the instruction set architecture (ISA) of the processor. While its not central to Cog I expect we will be improving the FFI as part of the project, but that those improvements would be across all platforms and some of them appear in the interpreter also. Fundamental will be source-leel compatibility for FFI calls.

HTH
Eliot

8. **Sandro** | 07-Jun-08 at 10:16 am | Permalink

Is your JIT based on an existing project, like GNU Lightning? If not, I don't suppose the JIT could be made a shared library that others could use? Pretty please? 🙂

———

Hi Sandro,

that's a very interesting and important topic that I'll probably post on more than once. There are at least two levels at which this is important. One is in the JIT's front-end in the interface for bytecoded methods. I want Cog to be easily reconfigurable for different bytecode sets, a range of object models (even if it'll be a limited range) and different primitive sets. The other is in the JIT's back-end and how it produces machine code specific to a particular ISA.

Ian Piumarta has done a nice library called ccg which might be useful. Essentially one can define an abstract machine very close to typical current 32-bit or 64-bit processors and have the JIT generate instructions for this abstract machine. In some way the abstract machine instructions get converted to the actual machine code of the target processor. This can be as crude as defining C macros. Bit I think there's advantages, e.g. in branch generation, of *moving up one level where an abstract instruction is a C struct with a number of fields, input registers, output registers, opcode, etc. With this level of abstraction I think issues such as instruction reordering, compact literal encoding and branch generation are easier to manage. Again more detail in a subsequent post.*

However, I don't think it'll be as simple as a shared library. Instead it'll be a kit of source components that you can configure and compile for your system. At least that's the hope. I very probably won't be able to do things like write a Ruby JIT but I'd love to see other people take Cog and do that. And I'd love to be able to do that fast enough to stay relevant 😉

Eliot

9. **Giovanni Corriga** | 07-Jun-08 at 2:12 pm | Permalink

Hi Eliot, do you think you work can integrate with Bryce's Exupery

———

Hi Giovanni,

alas no. I think the two aproaches are antithetical. Exupery is in the tradition of an Piumarta's 68000 compiler he did for his thesis and Justin Graver's and Ralph Johnson's TypedSmalltalk. These systems have platform-specific code generators at the image level which output ISA-specific machine code. Cog will be in the tradition of PS and HPS, Peter Deutsch's Smalltalk JITs which have image-level compilers that produce bytecode and VM-level code generators that take bytecode as source and produce ISA-specific code, but hide it from the programmer.

I'm a big fan of this latter architecture. It produces reasonably efficient code (see how VisualWorks stacks up on the computer language shootouts), it preserves the investment in and simplicity of the existing bytecode-centric Smalltalk compiler/debugger/browser architecture, and iis one important factor in platform-independence and image portability (the FFI and how OS facilities are accessed being the other). But an even stronger reason is that I think the level of abstraction introduced by bytecodes leads to a more modular and maintainable and hence reliable system. At least that's my prejudice.

I think an in-image native code compiler temps the system implementor with the prospect of high-performance at the cost of sins like in-lining, decorating source with type declarations, and so on. A Faustian bargain. Whereas, IMO, keeping the separation is not an obstacle for ultimate performance. I think that with the appropriate abstractions the hybrid bytecode/machine-code *architecture can produce code very close to that achieved by high-performance static compilers while retaining platform-independent at the bytecode level. But this remains to be proven and is a long ways off. There are more immediate concerns* 🙂

Eliot

10. **David** | 07-Jun-08 at 2:18 pm | Permalink

    Eliot – Nicely written. I didn't know about Sheherazard or the "Smalltalk" vs Greek gods thing.

    David

    ———

    Thank you David,

    Sheherezade is she of the 1001 nights (follow the link) who saved herself from slaughter by telling great stories, one a night. I hope to keep your interest by keeping the Cog blog interesting 🙂

    "Smalltalk" vs the Greek gods is about why Alan Kay chose to name Smalltalk Smalltalk instead of Thor or Zeus or any of the other names popular at the time. Alas his excellent article in History of Programming Languages isn't available and so I posted a link to a generic google search which throws up some secondary accounts. If you can track down Alan's original I recommend it.

    Cheers
    Eliot

11. **Damien Pollet** | 08-Jun-08 at 4:27 am | Permalink

    I know you don't hold a grudge against the NewCompiler, I'm worrying essentially about feature duplication and community fragmentation.
    BTW, what about coming to see us at ESUG in Amsterdam?

    I would love to! I'll explore this with … the buro 🙂

    (in follow-up to Giovanni)
    Exupery compiles byte-code to native, so it's clearly a different layer than the "high-level" compiler. The main difference I see is that since it lives in the image the code can easily be read or changed without building a new VM each time. If the Cog JIT lives in the VM, I suppose it will be C or Slang? Are those nice languages to write a compiler in?

    *C is not too bad and I'm very familiar with it, but Slang is C minus the few abstraction facilities that C provides (like bitfields in structs) and so in my experience is **not** nice to write in. I think it better that the JIT be in the VM in C. But it would not be an optimising JIT. Please wait for an upcoming post on "AOStA" which is how I'd like to do adaptive optimization at the image level. Things should become clear then.*

Also at which level would optimizations (like, say, trace trees) be implemented? Can those work on bytecodes so the JIT remains as a simple as possible?

Yes exactly. The JIT should just be a code generator. An adaptive optimizer can be entirely in Smalltalk, can analyse bytecoded methods and the state of in-line caches through Contexts and produce new optimized bytecoded methods that the JIT translates into machine code.

Best
Eliot

12. **Sandro** | 08-Jun-08 at 10:24 am | Permalink

> Ian Piumarta has done a nice library called ccg which might be useful.

GNU Lightning was built from the concepts in Ian's ccg IIRC.

> Essentially one can define an abstract machine very close to typical current 32-bit or 64-bit processors and have the JIT generate instructions for this abstract machine.

Yes, this is similar to what I was referring to. I don't see why this couldn't be done as a shared C library, assuming the JIT is written in C. As you say described, the JIT backend is just for some abstract machine, so it should theoretically be fairly reusable. Unless you feel there's some downside to this approach that I'm not aware of.

> However, I don't think it'll be as simple as a shared library. Instead it'll be a kit of source components that you can configure and compile for your system.

I see. I was wondering if it would be a more complete alternative to libjit used in the DotGNU project, but with the more liberal license you mentioned.

Let me take a look at libjit. When I look at things like LLVM what I see is too low level a system. It manages code generation but I don't see facilities for in-line cache management, polymorphic in-line cache generation, stack frame management, code zone management, etc, which are facilities that I think one needs. At the same time I don't quite know how to configure a complex code generator that would provide an interface at the level of bytecoded methods that could efficiently be used as a shared library. One would have a lot of configuration of the bytecode set, method format etc on loading the library. Whereas setting up a system of source components should be much more straight-forward. If that makes sense?

Eliot

13. **Florin Mateoc** | 08-Jun-08 at 12:21 pm | Permalink

Hi Eliot,

Hi Florin!

This is wonderful news!

So Smalltalk may still have a chance in this brave new world of open-source dynamic languages…

Please allow me to be jump in with some feature requests, while not too many things are set in stone:

1. vm support for breakpoints – VA does it nicely, and this allows the image-side to be kept clean and simple. I liked the functionality brought by PDP in VW, but I hated what it did to the parser and debugger

This is a great idea. Feel free to post sketches or a specification etc.

2. vm protection against stack overflows (infinite recursions)

yes. I guess you mean more than a low space warning? Perhaps a per-process maximum depth that is only tested when a stack page's worth of activations is evacuated to the heap to make room for more activations? Again expound, expound 🙂

3. bytecodes for fast and guaranteed access to an object's public and hidden fields (size, class, and identity hash). These can be expressed in source code with what you already have (e.g. in VW):

thisContext _objectClass: anObject

and so on, but it does not have to be translated to normal bytecodes/message sends, as this is one of the few instances where things can be statically compiled (if only we had the right bytecodes)

Right-oh. I call these Mirror Primitives after David and Gilad's work on Self.

4. vm support for block iteration (VA's #apply:from:to: and its relatives), which allows for non-bounds-checked access inside the iteration. I know that this would not always behave "correctly", e.g. the following fails with an index out of bounds in VW but succeeds in VA:

```
|a |
a := Array with: 1 with: 2.
a do: [:e | a become: (Array with: 3)].
```

But I don't know off the top of my head of any situations where somebody would legitimately want to use such code, since any such example would imply modifying a collection while iterating over it. Actually, I just tried a slightly modified version:

```
|a |
a := Array with: 1 with: 2.
a do: [:e |
```

Transcript cr; show: e printString.
a become: (Array with: 3)].

and this one blows the VA image up, which is definitely not nice, but again, there is something to be said about the fact that VA has lived with this "bug" for so long and nobody seems to have complained

Hmm, I think bytecode-to-bytecode adaptive optimization is a more general approach. See an upcoming post on AOStA (Adaptive Optimizing Smalltalk Architecture).

5. This is more minor, but for the performance obsessed among us, VA has an interesting couple of primitives for (identity) hashed collections:
#basicHashOf:range:
and
#indexOfNextNilOrIdenticalTo:startingAt:
They avoid some computations plus any iteration on the image side when doing hashed lookup

Again my hope is that image-level bytecode-to-bytecode adaptive optimization renders the need for go-faster primitives obsolete. The image-level optimizer plus the JIT should be able to generate code with equivalent performance from the Smalltalk bytecode.

Best
Eliot

14. **Sandro** | 09-Jun-08 at 7:28 am | Permalink

> [LLVM] manages code generation but I don't see facilities for in-line cache management, polymorphic in-line cache generation, stack frame management, code zone management, etc, which are facilities that I think one needs.

Right, LLVM is a compiler backend, not a VM runtime. The LLVM devs would say that you should express your polymorphic inline caches as a program in LLVM's IR language. Explicit stack frame management is achieved by a type of CPS conversion. The whole point with LLVM is that it provides such extensive optimizations and such good code gen, that many custom optimizations you might provide in your VM are obsolete. It carries a heavy price though: LLVM will add about 1MB to your VM.

I had the chance to visit Apple's WWDC last week and bumped into Chris Lattner, one of LLVM's authors, now at Apple. We had a good conversation. The consensus was that LLVM is appropriate when there are large basic blocks to optimize. In the initial release of the Cog JIT (scheduled for April of next year) there won't be large basic blocks. Typical Smalltalk methods execute about 6 bytecodes between sends which means that there really isn't *much for an optimizer to work on. A simple native code generator can generate sufficiently high-quality code for this kind of instruction mix in much shorter time with less run-time overhead than LLVM.*

However, if Cog continues to evolve and we're able to do adaptive optimization (or "speculative in-lining", as Chris calls it, a name I like) then the VM should be able to present LLVM with large basic blocks at which point LLVM becomes a much more attractive back-end.

Some necessary VM-specific operations might be provided as LLVM "instrinsics".

> At the same time I don't quite know how to configure a complex code generator that would provide an interface at the level of bytecoded methods that could efficiently be used as a shared library.

An approach similar to LLVM's, where the codegen is abstracted via a low-level intermediate language, or an approach similar to an typed/untyped assembly language for an abstract machine, like GNU Lightning, would suffice. In other words:

```
typedef _codebuffer code_buffer;
typedef int ireg;
typedef int freg;

// dest = i + j, set overflow
void iaddr_t(ireg dest, ireg i, ireg j, code_buffer c);
// dest = i – j, set overflow
void isubr_t(ireg dest, ireg i, ireg j, code_buffer c);
…
// the rest of an assembly language api; you can also expose continuation capture, and other low-level VM constructs via such an API.
```

Your JIT sounds attractive since you're going to support more platforms than GNU Lightning, so if it could be extracted into a library, it would be useful for many projects (one of mine included! ;-). Mono would have been good here too, and I tried contacting the Mono devs about this, but received no response.

I hope you'll be able to use it!! However, "library" is one thing and "source kit" something else. I expect to provide a "source kit" which is easy to reconfigure for your project, rather than a library one loads and uses through a well-defined API. Part of that is reducing design effort on my part, but part of it is my experience in VMs telling me that the domain is too complex to nail down with a comprehensible API and still produce fast code. I'm probably wrong about this, not I'm not the guy you want to try to design the API. That's not my forté.

As for the abstract assembly language, which I think is the more flexible option, I've considered two approaches: stack machine and register machine. Using a register machine either means that the assembly is arch-specific in order to take advantage of all the registers, or it exposes a common subset register file like GNU Lightning, so some registers may go unused. This is the advantage of the the stack machine, which when combined with a transformation from arch-independent stack code to arch-specific register code [1], can be quite attractive. The transformation seems quite expensive though, so I'm waffling a bit here. An SSA-based intermediate language, like LLVM's, is a good compromise here.

[1] Virtual Machine Showdown: Stack Versus Registers, http://www.sagecertification.org/events/vee05/full_papers/p153-yunhe.pdf

Very interesting; thanks! I'm (happily) stuck with a stack machine with Cog. Smalltalk has (almost) always been a stack-based language centered around bytecodes. The transformation from stack to register based code in a JIT is rather straight-forward. Ian Piumarta's thesis on Deferred Code Generation presents the idea. Peter Deutsch's HPS VM does it (and may have done it first), as do some Java VMs:

Ali-Reza Adl-Tabatabai, Michael Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M. Stichnoth, "Fast, Effective Code Generation in a Just-In-Time Java Compiler", In Proceedings of the SIGPLAN `98 Conference on Programming Language Design and Implementation, p. 280-290. Published as SIGPLAN Notices 33(5), May 1998.

So Cog will stick with stack based code. I expect really high performance to come more from adaptive optimization/speculative in-lining and mapping floating-point values to FPU registers than from the virtual instruction set. But we can discuss this more in subsequent posts.

Cheers!
Eliot

15. **Florin Mateoc** | 10-Jun-08 at 12:02 am | Permalink

This is a reply to your embedded comments to my comment, but I don't know how to thread it/embed it.

1 (debug support)

In VA, Process implements #debugInterrupt (it opens a debugger on the sender frame), which is sent by the VM. This looks like an OS-level debug interrupt, not related to image-level breakpoints. Nevertheless, it could come in handy. Come to think of it, the profiler could also use a notification mechanism from OS instrumentation probes.

Also in VA, Process has some settable debug bits: #breakEveryBytecode, #breakOnEnter, #breakOnExit. The debugger uses them and resumes the debugged process at every step, thus avoiding (image-side) simulation.

As for the VM-side breakpoints, upon closer inspection, VA does not actually keep things in the VM, it uses DbgBreakpointedMethod(s), with inserted break bytecodes, which is pretty similar to ProbedMethod(s) in VW. But they do use a VM-originated message send when hitting the breakpoint (DbgBreakPoint class>>breakpointEncountered:), so using this as a starting point:

ProcessorScheduler should implement #breakpointEncountered: (with the breakpoint number as an argument), which will be sent by the VM. On the image side, the breakpoint number will be mapped to a breakpoint instance.

When setting a breakpoint, a new breakpoint instance with a next available number is created. The target method source is parsed (to get the corresponding parse node index) and recompiled (to get the corresponding bytecode index). The breakpoint is assigned its associated parse node index. The VM is passed through a primitive the breakpoint number, the method and the bytecode index.

2. (stack overflow)

Yes, I mean testing the stack depth. I suppose this could be done upon some allocation failures. I would guess that they do occur whenever we have infinite recursion, and they might be in a slower code path. And, somewhat related, the stack depth could even be exposed as a primitive (e.g. the profiler could use that)

Cheers,
Florin

16. **Armando Blancas** | 10-Jun-08 at 11:13 am | Permalink

Really cool project; I'm glad to know you're doing this. It's been a while since I used your BrouHaHa back in '92 at QMW. Now, good professor, my 8 year old is doing fine with Scratch and I plan to move him up to Squeak and Croquet when he's ready –on a much improved VM, looks like. Good luck.

17. **Jecel Assumpção Jr** | 13-Jun-08 at 3:15 pm | Permalink

Very cool project! I have designed a few bytecodes with a prefix instruction. Since these were meant for hardware implementation the literals were mixed with the instructions (duplicate literals are not as common in Self-like implementations as in Smalltalk-80) but perhaps some of the ideas might be useful to you.

Thanks, Jecel.

yes, I did prefixes in VisualWorks along with Steve Dahl (he the bytecode compiler me the VM). It works well. With short-form and long-form bytecodes the prefixes don't have to apply to the short forms, only the long-forms. So the overhead is reduced. I'll be presenting a prefix-based bytecode set sometime, probably in a few weeks.

Eliot

18. tim@rowledge.org | 17-Jun-08 at 1:15 pm | Permalink

Hi there-
so nice to see that you've got a chance to *actually* improve the squeak vm. After a decade of attempts to get something moving I was about to just give up and go do something else. Now perhaps it's worth hanging around to see if I can help or heckle to some effect.

I'd like to urge that you dump the stupid old compiled method format. After all, if you're anticipating changing the definition of some bytecode and adding closures properly (at last!) tthen the image really isn't going to be compatible with older vms. Besides, the current CM is total crap. It's a bad idea from top to bottom and the hacks added in recent times, such as the properties array forced into the literals which are in turn etc etc just make it worse and worse. Clean it out! Traits broke the format as well, so far as the VM is concerned. So please, lets' fix it properly and make the GC code simpler as well. Your great grandchildren will thank you.

Many years ago, Ian P was very excited about the performance enhancing possibilities of a cleaner CM design, claiming that having the bytecodes and literals as separate arrays would drastically simplify a translator. I'm not entirely sure I understood his arguments and of course they never had any practical value because he never finished anything that we could look at.

I think something like
Object subclass: #DecentCompiledMethod
instanceVariableNames: 'header bytecodes literals properties'
classVariableNames: ' '
poolDictionaries: ''
category: 'Kernel-Methods'
would be nearer the mark. I mean, just look at the current implementation of #methodClass. Or that old crap of fileIndex etc. Blech. Kill it.

I think this is a problem of not seeing the wood for the trees 🙂 The problem is not the format of CompiledMethod it is not being able to change it or subclass it. But these limitations, not being able to add inst vars to CompiledMethod and not being able to subclass CompiledMethod are restrictions that only apply in the 16-bit BlueBook VM. The current

Squeak VM is more than cabable of allowing CompiledMethod to be subclassed and to add inst vars to CompiledMethod and subclasses. It'll take a bit of work in the ClassBuilder but no work in the VM. I have a post prepared on this but it is in the queue behind the next Closures post which is itself held up by my not being able (yet) to post the code. I don't want the posts to get too bogged down in detail so they're not the avenue through which to publish code (no one's going to copy/paste code out of a blog post anyway!). Perhaps I should just post the message out of sequence.

Were you anticipating doing the translation under the covers within the vm as hps, or within the image? Or some other clever hack? I quite like the in-image idea as a general concept since it allows for a lot of flexibility. For example Bryce has had quite a lot of success writing a compiler that runs in the background, which is the 'obvious' approach. I'd like to see if having a remote compiler machine that you can send the method details to and get back a finished object might work. A local server image, or a remote one, or one that caches and can return almost immediately, or even one that submits it to Amazon's mechanical turk could be tried. Another option would be to implement the translation as whatever needed mix of image code and plugin primitives; that would make it very easy to have a flexible system configurable at run time.

I like keeping as much stuff in the image as possible. But there are abstraction boundaries to maintain to keep a clean design. One important one is that of contexts. Contexts are a superb abstraction for activations, easing the implementation of processes, the debugger and so on. But they suck as a form for real execution. So contexts belong in the image and clever tricks to do without contexts most of the time belong in the VM. But that implies that any code to do with hiding contexts belongs in the VM which implies that the lower levels of the code generator are in the VM.

I favour an architecture that has an adaptive optimiser/speculative inliner up in the image which analyses bytecoded methods and execution state through contexts and which creates new bytecoded methods which inline others. In this architecture the optimizer targets portable bytecode and the VM's code generator still has the responsibility of converting this to a particular machine code and of mapping back from stack frames and machine code PCs to contexts and bytecode PCs.

I also don't believe that code generation is so slow that one would ever benefit from cacheing generated code. In fact I think that would slow things down. Note that the in-image adaptive ptimizer has no problem cacheing optimized methods up in the *image, and that generaign efficient code from longer optimized methods in the VM is still something that could happen relatively quickly given a cheap and dirty register allocator. My hunch is that one might be a factor of two slower than a slower aggressive optimizer but one wouldn't be so far away that you'd throw up your hands in disgust and go back to a static language. One only has to do significantly better than the state-of-the-art in dynamic language VMs to add value. One doesn't have to beat fortran.*

Cheers!
Eliot

tim
—
tim Rowledge; tim@rowledge.org;
http://www.rowledge.org/tim
Useful random insult:- One chicken short of a henhouse.

19. **John M McIntosh** | 17-Jun-08 at 5:49 pm | Permalink

    >A local server image, or a remote one, or one that caches and can return almost immediately, or even one that submits it to Amazon's mechanical turk could be tried.

    Oooh. Mmm I wonder caching at the central server? I wonder if you send hashs of the method if it could send back the cached compiled code, otherwise you send the code and get back compiled version. Obviously you can decide between using a common pool, or an agreed upon encryption for private usage.

    Then for base squeak you could pre-load cached compiled code at startup time since global usage dictates a certain pattern

    *There is *no way* in which copying generated code over the network is going to beat generating fast code in a VM on the fly. No *way*. Have you looked at start-up times for VisualWorks? Note that even though the images Isaac Gouy runs on his shootout site take about 0.2 seconds to start-up that that time is dominated by walking the heap doing things like voiding old graphics handles, not in generating code. At least last time I looked compile time wasn't the dominating factor. JITs can be very fast. Peter Deutsch's figures were always around 20 machine instructions executed to generate a machine instruction. Even at one two or three orders of magnitude worse than that you're going to slaughter copying code around the network. Fugedabahtid.*

    Eliot

20. **tim@rowledge.org** | 17-Jun-08 at 10:52 pm | Permalink

    Well, here we hit the complete inability of blog layout to support sensible back and forth debate.

I’m going to try to comment on Eliot’s comments o my comments without getting us all lost. GPS doesn’t work in here so good luck….

WRT CM format
“I think this is a problem of not seeing the wood for the trees The problem is not the format of CompiledMethod it is not being able to change it or subclass it”
Interesting point. Sure, let’s make it possible to add instvars to byte objects and word objects. Let’s add complexity to the classbuilder and debugger/inspector to support it. Is that actually simpler (remember you claimed to like TSTTCPW) than using the ‘normal’ class structures we already have? It would take some god arguments to convince me that it is so.

WRT ‘in-vm’ JIT or ‘in-mage’
I’m pretty sure that this –
“One important one is that of contexts. Contexts are a superb abstraction for activations, easing the implementation of processes, the debugger and so on. But they suck as a form for real execution”
does not actually logically imply this –
“So contexts belong in the image and clever tricks to do without contexts most of the time belong in the VM. But that implies that any code to do with hiding contexts belongs in the VM which implies that the lower levels of the code generator are in the VM.”
I don’t see why an in-image translator shouldn’t be allowed knowledge of the context handling, just as it is obviously allowed knowledge of the machine architecture. It doesn’t impact the debugger any more than an in-vm translator since you wouldn’t be debugging the generated code.
Further, one potentially very useful form of in-image translator would simply pass the relevant objects to a primitive (pretty much the same code as one might have in an in-vm translator) BUT it would allow the policy of what gets translated and when to be handled in the image. This might be of benefit in not ‘wasting’ time on translating code that is only run once, or it allow a choice between a quick-jit and a smart-jit at need. Or it might fail the prim completely on machines where there is no translator plugin yet provided or where the writable memory available on the machine is too small.
Having a separate machine providing translated results does not necessarily mean going over a network connection – it could be another thread on the same machine. Perhaps that would be a way of getting an aggregate speed boost from many-core systems – and of course one could do that with an in-vm translator too.
On some restricted systems it might be that performance or memory limitations prevent the system from being able to run a good translator and passing the work across a network is the better

option. And on some other forms of restricted system it might be that there is read-only memory available that could be used to store cached translated code to some benefit.

Just offering some different views from odd angles.

tim

21. **richie** | 20-Jun-08 at 9:14 am | Permalink

Eliot, this is great!!!

I wish I could be doing it too 🙂

Hi Richie,

I certainly hope there will be room for collaborators. I have to get my act together with a code repository and a bootstrap for a non-Qwaq image. Once this is done others will be able to get at the code.

Anyway, two comments:

When you say you are going to stick to a stack interface, I guess this is not 100% strict, and you are willing to go for tradeoffs such as passing receiver (and maibe first argument) in registers (eax and edx maybe?). Answers also in registers (maybe eax), and holding a reference to self in some other register (esi sounds good)? I have not done any performance test, but I think we don't need to do them to demonstrate this gives a great performance boost, and IMHO it doesn't break the stack abstraction too much (you only need to think that the top of the stack is held in eax 🙂

OK, I'll go into this in more detail in another post but this is a most important point. The "interface" to the VM is still contexts. Internally the VM uses a stack for efficiency but this is (almost) completely transparent to the Smalltalk programmer who still sees only contexts. See this for a poorly written account of the details.

The calling convention will probably use Peter Deutsch's convention for HPS which passed the receiver and last two arguments in registers. Using the last two makes the code generator simpler. This means that primitives like at: and at:put: don't even touch the stack. The registers get pushed on method entry proper with a callee-saves convention.

On a different side, security is a huge issue, specially when it comes to JITs. For example, a very easy way to break out of the VM on a JIT is to unbalance the stack (when real and VM stacks are shared), if you push a constant and end the method, the return will just jump to the constant. This is a very clear case when arbitrary native code could be executed. Of course, if you are not thinking about security at all, and there's not going to be no sandboxing, maybe the same could be done just injecting bytecodes.

Bytecode verifiers are one way to constrain this type of attacks, but they are not always enough.

I have some experience in security, and I'm willing to follow up on this if you are interested.

I'm very interested. Please do. Again much more details when I post on context-to-stack mapping, but I can say a little about the stack organization now. The VM's Smalltalk stack will be housed on the C stack (created by alloca) but this is effectively sandboxed. One can only run bytecoded methods on the Smalltalk stack, bytecoded methods have finite stack, and calls on the Smalltalk stack are bounds checked. So I don't think its possible to violate security through bytecode.

I'm interested in Gilad Bracha's and David Simmons' ideas for security and would like Cog to be used to implement a secure language (perhaps Newspeak) as well as "vanilla" Smalltalk. Gilad's security model in Newspeak is based on the bytecode set being secure by design, so that for example a private send bytecode has an implicit receiver so one can't create bytecode that pushes some arbitrary receiver and does a private send to it. Since I want to make Cog easily configurable w.r.t. bytecode set I hope it'll be easy to use it for e.g. Newspeak.

Eliot

22. **richie** | 24-Jun-08 at 6:13 am | Permalink

I'll lookup Gilad Bracha's and David Simmons' ideas on security. Do you know if there's any virtual machine (with JIT) implementing any of this ideas? I'm more familiar with Visual Smalltalks VM/JIT, but willing to take a look at any other.

Yes. I'm pretty sure David's S# and SmallScript VMs implement all his security mechanisms. Contact him for details. He's on LinkedIn for example.

I've only given a quick diagonal read to your paper on "Context Management in VisualWorks 5i", and it looks similar to what VisualSmalltalk does (not that I know all VS' details).

Do you have a reference? I was under the impression that what I'd done for the context proxy mechanism was original. Does VS even have contexts?

One thing I can say is that in VisualSmalltalk is easy to demonstrate what I mean:

(Object >> #boom) byteCodeArray: #[2 19 72]

2 NoFrameProlog
19 PushSmallInteger0
72 Return

This will jump to native memory address 1 (SmallInteger 0). Manipulating this, an attacker can execute arbitrary native code, and that's game over.

Right. But that requires that NoFrameProlog is in the instruction set. By bounds checking jumps at JIT-compile time and not providing instructions which allow one to jump to or return to arbitrary integers the bytecode can prevent executing arbitrary code. For that one would have to use the FFI 🙂

This is an example of the kind of native attacks on JITs that are most dangerous.

As I said, I still need to read your paper (have it on my loved kindle now :), and the other's you referenced.

23. **richie** | 29-Jun-08 at 11:59 am | Permalink

On VisualSmalltalk and Contexts:

I don't really know the details. Apparently there is no thisContext in VisualSmalltalk, but you can somehow access the contexts stack using Process, although it's not the same.

However, they do have something like what you describe to cope with BlockClosures:

The prologue of a nativized method somehow depends on the Blocks in the method, and there are some cases where the prologue includes an Array creation, and in this cases, locals and temps are accessed with a different set of Bytecodes: LoadContextTemporary/PushContextTemporary/StoreContextTemporary

This newly created Array referencing to context slots is somehow called "environment temporaries". You can see the logic of how it's used (at compile time) in ScriptNode >> #rebindTemporaries, at some point it reads:

```
info := tempScope bindingFor: t value ifNone: [].
info isExternallyReferenced
ifTrue: [
t
binding: (self environmentTemporaryBindingClass
new
name: t value
position: self incEnvTemporaryCount).
t binding markReferenced]
ifFalse: [
t
binding: (self stackTemporaryBindingClass new
name: t value
position: self incStackTemporaryCount)].
```

(how do you do nicely formatted posts on this blog?)

Most of this is guess work, as you can imagine.

On the security issues:

Using NoFramePrologue was just an example. I'll give you another example now, but the key point is: Security problems can be all over the place, from design down to implementation. Security in VMs is usually dealt with through sandboxing of some sort, but when it comes to a JIT, there are more problems: basically, if an attacker can control VM Bytecodes, he can somehow control what native code is emitted, and, unless specific validation is made, it's quite possible the attacker can escape the VM.

So, another example, from VisualSmalltalk, would be:

(Object >> #methodDictionaryArray:) byteCodeArray: #[16r50 16r9D 0 16r48]

Bytecodes:

16r50 LoadArgument1
16r9D StoreInstanceN 0
16r48 Return

You can use this method to change the methodDictionaryArray of any object. It works because instance variables are indexes as 1-based, hence StoreInstanceN 0, stores at instance variable 0, which is just before the first instance variable, i.e. the methodDictionaryArray of the Object.

In VisualSmalltalk there's already a primitive (97) to do this (which could be protected through some sort of Sandboxing). However, this not only breaks the integrity of objects, it's even worst (tested):

16r12345678 methodDictionaryArray: 'hola amigos!'

will write a pointer to the string 'hola amigos' at absolute memory position (16r12345678*2+1) (*2+1 is to convert from SmallInteger to native representation of SmallInteger, marking it with the lowest bit in 1, similar to VisualWorks, but I think VW uses 2 bits instead of 1).

So, here's another example. I could probably find more, but it'll still be for VisualSmalltalk. If you are interested, I could take a look at anything else, but it'll probably take me some time to find something.

Sorry for the long post 🙁

24. **richie** | 29-Jun-08 at 1:23 pm | Permalink

An excelent paper on JIT nativizers security flaws by the LSD team can be found in http://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-lsd-article.pdf. Although it's old (2002), it's still current.

25. **Paolo Bonzini** | 14-Jul-08 at 2:00 am | Permalink

As you said, speculatively inlined methods generate larger basic blocks. However, even simpler JITs (e.g. using the code generation techniques in Ian Piumarta's Ph.D. thesis) do generate large *superblocks*. I wonder if LLVM has a pass like GCC's tracer, which produces large basic blocks from large superblocks…

26. **Paolo Bonzini** | 14-Jul-08 at 2:07 am | Permalink

@richie: one of the nice things in a dynamically-typed language like Smalltalk, without any primitive typess, is that a bytecode verifier becomes a very simple thing. GNU Smalltalk has one, it's around 700 lines of C code (compared to 3200 lines of C++ for the Java verifier).

A verifier would catch the cases you proposed here.

27. **Kirill Kononenko** | 17-Apr-09 at 1:05 am | Permalink

Hi

I suggest use libJIT instead of LLVM overkill. For example, these are just a couple of things which LLVM does not support for .NET and Microsoft Common Intermediate Language implementation:

* the whole spectrum of ECMA 335 for CLR types and operations
* async exception handling
* precise stack marking
* multiple custom call conventions

All these things are supported in libJIT, as the Portable.NET JIT has been developed with libJIT. Also libJIT is used in HornetsEye for digital video and image processing with Ruby!

See also why LLVM is an overkill here:
http://lists.ximian.com/pipermail/mono-devel-list/2009-April/031640.html

http://lists.gnu.org/archive/html/dotgnu-libjit/2004-05/index.html

http://code.google.com/p/libjit-linear-scan-register-allocator/

Thanks,
Kirill

28. **Eliot Miranda** | 17-Apr-09 at 9:57 am | Permalink

Hi Kirill,

interesting comment! I too think that LLVM is overkill. Having worked for nearly a year in Slang, the Smalltalk development system for the Squeak VM, I really like the experience of developing in a high-level language, so I really want my JIT written in Smalltalk. Recently I met Slava Pestov at the PyCon dynamic VM summit. Slava has written factor, a hybrid of a higher-order functional language with forth. He has developed a really cool jit framework entirely in a high-level language, and it is self-hosting in that one can make modifications to the jit as one is running.

So right now my thought is to write a factor-to-Smalltalk translator and port the factor framework to Smalltalk.

What is libJIT written in? How is it developed (tools, etc)?

29. **Kirill Kononenko** | 21-Apr-09 at 1:22 pm | Permalink

Hi Eliot,

libJIT is written in C. There is a good documentation her from Rhys Weatherely the original author of libJIT:
http://www.gnu.org/software/dotgnu/libjit-doc/libjit_1.html
http://www.gnu.org/software/dotgnu/libjit-doc/libjit_3.html#SEC5
”

The primary interface is in C, for maximal reusability. Class interfaces are available for programmers who prefer C++.
Designed for portability to all major 32-bit and 64-bit platforms.
Simple three-address API for library users, but opaque enough that other representations can be used inside the library in future without affecting existing users.
Up-front or on-demand compilation of any function.
In-built support to re-compile functions with greater optimization, automatically redirecting previous callers to the new version.
Fallback interpreter for running code on platforms that don’t have a native code generator yet. This reduces the need for programmers to write their own interpreters for such platforms.
Arithmetic, bitwise, conversion, and comparison operators for 8-bit, 16-bit, 32-bit, or 64-bit integer types; and 32-bit, 64-bit, or longer floating point types. Includes overflow detecting arithmetic for integer types.
Large set of mathematical and trigonometric operations (sqrt, sin, cos, min, abs, etc) for inlining floating-point library functions.
Simplified type layout and exception handling mechanisms, upon which a variety of different object models can be built.
Support for nested functions, able to access their parent’s local variables (for implementing Pascal-style languages).
”

For example, if one wants to multiply two numbers:
int compile_mul(jit_function_t function)
{
jit_value_t x, y, z;
jit_value_t temp1, temp2;

x = jit_value_get_param(function, 0);
y = jit_value_get_param(function, 1);

temp1 = jit_insn_mul(function, x, y);

```
jit_insn_return(function, temp2);
return 1;
}
```

This will generate a body of this JIT. Then to compile it use jit_function_compile and run it jit_function_apply

30. **Kirill Kononenko** | 21-Apr-09 at 1:24 pm | Permalink

    One only needs to link with libJIT library to us its features.

31. **Kirill Kononenko** | 21-Apr-09 at 1:24 pm | Permalink

    One only needs to link with libJIT library to use its features.

32. **Kirill Kononenko** | 21-Apr-09 at 1:35 pm | Permalink

    I think libJIT is a really cool library.
    All the docs are by Rhys Weatherley at this time:

    http://www.gnu.org/software/dotgnu/libjit-doc/libjit_3.html#SEC5

    This documentation is really great. I can recommend it to everyone.

33. **Alain91** | 14-Jun-11 at 9:12 am | Permalink

    Hi eliot,
    congratulation for this amazing product.
    Is there any plan to include utf-8 character encoding, directly in the VM ?

34. **Eliot Miranda** | 15-Jun-11 at 11:05 am | Permalink

    Thank you, Alain. What exactly do you mean by supporting UTF-8 in the VM? Do you mean in pathnames in the file system primitives, which I think is already there? Do you mean better Character support in the Squeak image, such as VisualWorks' immediate Character class, which I do plan to add? Or do you mean something else?

35. **Alain91** | 17-Jun-11 at 10:56 am | Permalink

    I use pharo and seaside and it seems the utf-8 characters are decoded or encoded in smalltalk methods.
    As utf-8 is now a standard, for performance reason, I think about core primitives in the VM to encode and decode utf-8 character sets.

36. **Eliot Miranda** | 17-Jun-11 at 2:14 pm | Permalink

    Hi Alain,

I think putting effort into a better object representation, that will provide immediate characters, slightly faster message sending and significant;y faster at: and at:put: implementations will go a long way to making the Smalltalk UTF-8 encoding code faster and be more general. So rather than add primitives to the VM for specific VM cases I'd like to put my effort into increasing Smalltalk performance across the board.

cheers
Eliot

37. **Alain91** | 18-Jun-11 at 1:56 am | Permalink

Thank you for this kindly answer. Go ahead that's fine.

Post a Comment

Your email is *never* published nor shared. Required fields are marked *

Name		*
Email		*
Website		
Message		

Post

CLOSURES PART I »