GNU Jitter

a generator of efficient language virtual machines
for version 0.9.274.1-fa96-dirty, updated in December 2021

Luca Saiu

Table of Contents

Introduction

GNU Jitter is a software automatically building a portable, very efficient language virtual machine with performance close to native code, starting from a relatively high-level *specification* provided by the user. The VM state may include registers, stacks or a combination of both along with any runtime structure defined by user code; whenever possible the generator attempts to map such state data structures into hardware machine registers. The specification contains some C code associated to every VM instruction; the generator takes care of combining such small blocks of C into a whole body of code with low—usually zero—dispatching and branching overhead. The generated code includes a simple C API to dynamically emit and execute VM code, and a self-contained driver program for running VM routines from text files.

The generated C code is heavily conditionalized and can be configured to run using different dispatching techniques, of varying sophistication; the most efficient dispatching techniques rely on some architecture-specific—but *not* VM-specific—assembly support already included in this software; every dispatch but one also relies on GNU C extensions. As a fallback case, in the interest of portability, one dispatching technique is provided, `switch` *dispatching*, requiring nothing more than standard C.
Configuration parameters are transparent with respect to the VM semantics and even the C API: a VM routine will always behave in the same way independently from the dispatching technique and other configuration parameters, the only observable difference being execution speed.

As long as they are willing to respect the license most free software maintainers whose projects include an interpreter of some kind should seriously consider replacing their execution engine with a VM generated by this software. A *Jittery VM* will easily surpass any interpreter in performance and should even be competitive against JITs, with the sources being much easier to maintain and portability coming for free.

Audience

Jitter is designed to generate production-quality language virtual machines, with a strong emphasis on performance. Despite some effort to keep the discussion accessible I cannot in good conscience present the software or its documentation as suitable to beginning programmers. Taking full advantage of Jitter requires proficiency with the C language, at least some understanding of assembly and an intuitive idea about the performance model of native code running on modern CPU micro-architectures, particularly with respect to instruction bandwidth and latency, out-of-order execution, branch target prediction and cache effects.

That said, everybody is welcome to try. With the caveat in [About language virtual machines], page 2, beginning compiler writers might use a generated virtual machine instead of native assembly as a portability layer, even without mastering every intricacy of the implementation.

History

GNU epsilon (see *The GNU epsilon Manual*) is a programming language and my own main project, still to be officially released as "stable", built on an extremely simple core language called ε_0 and augmented with powerful syntactic abstraction. I have been working on epsilon for many years, rewriting it from scratch at least five times in a strenuous effort to make it "right". Even if many details of the implementation can still use more than some fine tuning, I believe that at least its general architecture is now correct.

epsilon is a general-purpose language, combining very low-level control (ideally at a level finer than GNU C; certainly finer than standard C) with high-level features allowing the user to express and combine complex syntactic forms. Ultimately every piece of epsilon code, no matter how complicated, gets rewritten into ε_0 before execution or compilation. Interpreting ε_0 is easy, but very inefficient.

Even worse epsilon is written in itself: execution begins from a crude “bootstrap” ε_0 interpreter running through the definitions of every high-level form, as macros and expression rewrites, which eventually make the language usable by humans. This idea of incremental definition relies on an epsilon program modifying itself in the style currently popular in the “dynamic language” community: global procedure, macro and variable definitions are added or replaced *at run time*, in a way challenging the traditional idea of compilation. On the other hand, and differently from what happens with the fashionable “scripting languages”, epsilon programs usually reach a point in their execution after which they stop self-modifying and become suitable for traditional, efficient compilation to machine language. This final compilation of non-self-modifying programs after they are automatically rewritten into ε_0 is, unsurprisingly, easier than in other languages; but executing a program while still in its “dynamic” form has always been frustratingly slow. Bootstrapping takes minutes even on a fast machine and interactive use, if not properly painful, lacks the expected smooth *feel*, the occasional “embarrassing pause” at the end of a complex macro invokation reminding the user every time that the implementation has not been tuned quite in the right way.

Attempting to accelerate ε_0 interpretation I quickly put together a stack-based direct-threaded interpreter, with the intention of using it in place of the current abstract-syntax-tree interpreter. That implementation still lives on a git branch but I will not integrate it in the mainline: at only 4-6x its speedup was disappointing.

In order to experiment with fast interpreters I started a new threaded-code prototype independent from epsilon. Very soon I had the idea of *specialization* (see [Specialization internals], page 42), one of the few true innovations in this project along with patch-ins (see [Patch-ins], page 42), despite its simplicity; the results were promising, but again the simple combination of direct threading and specialization was not enough to achieve the speedup I was expecting. I found myself reading and re-reading scientific papers (see [Bibliography], page 49), and adding more sophisticated ideas to my prototype to the point where it is now debatable whether the software still counts as an *interpreter* generator.
Some ideas definitely belong more in JITs than in threaded interpreters, and in fact the most advanced dispatch supported by Jitter requires no threaded code to be processed at run time, nor indeed any data structure representing routine code other than an array of native machine instructions in executable memory—“interpreting” a routine means jumping to its first native instruction, from where execution continues.

Soon enough what I had planned as just a fun hack for a few weekends turned into something ambitious and useful for others, independent from epsilon which will now require it, but *without being required itself* for compiling or running Jitter.

[*FIXME*: **list some milestones**]

Jitter was officially accepted as part of the GNU Project on December 17th 2021.

About language virtual machines

Language virtual machines are fashionable: people like to compile to VMs for portability, and to reuse the same VM as a target for several different languages, forcing VM specifications on one hand to maintain backward compatibility, and on the other to provide some level of generality.

The VM-runtime design nowadays has gotten so pervasive to be almost the expected default, to the point that the word “unmanaged” was coined to describe a language runtime *not* depending on a virtual machine, as if directly generating physical machine instructions were some kind of oddity requiring justification.

I disapprove of this trend.

When *not* to use VMs

First of all the idea of compiling to a VM for portability relies on the presence of some runtime system, usually very large and either written in a low-level language, therefore simply shifting the portability problem, or mostly built on the VM language itself making programs less efficient than it would be otherwise possible, particularly at startup.
Moreover most languages need some form of compilation to machine code to attain an acceptable level of performance. JITs are now part of every language VM at least attempting to obtain good performance, but again they are not trivial to port, and nowadays most JITs will target just the one or two most popular hardware architectures.
I have grown skeptical of claims about language VMs as a solution for portability: if that were really the main concern then simply using C plus some reasonable tool for handling the occasional system oddity (GNU Autoconf and Gnulib come to mind) would provide for even better portability, and without compromises on performance.

A few VMs also provide features unrelated to portability, particularly sandboxing or some safety checking of programs at startup. Ideas such as the JVM's "bytecode verification", while scientifically interesting, are a mismatch for most use cases and should never get in the way of running compiled code efficiently.

Exposing a VM as a software API decoupled from a compiler is a source of inefficiency when independent projects with different needs target the same VM, or even when a single compiler needs to evolve but at the same time needs to retain compatibility with older versions of a VM satisfying a given specification; the impossibility of implementing efficient method tail calls in the JVM depends on such a case of excessive decoupling—and in some measure, I dare to say, on a community of programmers in general too conditioned by cultural inertia and irrational fashions to appreciate the benefit of a clean break.
[*FIXME*: **weaken: the typical use case of java may have changed**] Speaking of Java, the entire performance profile of the JVM stands out as a *spectacularly* poor fit for server-side web programming, the most popular use case for the language: processes are spawned often, live for a short time, compute very little and only consist in running local, statically compiled bytecode on one physical machine with a known architecture; bytecode is hardly ever exchanged over the network or generated at run time.

If Java is one case of incorrect application of VMs its example is not the only one, nor the worst. Soon after web browsers started to include good JITs for JavaScript—an accomplishment in itself for a language so difficult to execute efficiently—people started to use JavaScript as a compilation target for *lower*-level languages, usually restricting the generated code to the JavaScript subset that current web browsers can JIT-compile well. What would have been an entertaining hack turned into a grotesque perversion of huge machine-generated code with mediocre performance, and very troubling political implications: distributing important software as generated JavaScript code to run on a web browser over the network prevents its users from easily changing the source, introduces gratuitous dependencies on the network when none would be needed, and potentially exposes user data to the server or to eavesdroppers.
On the technical side some of these projects are very impressive: I have seen the old Doom game run on a web browser at amazing speed, the frame rate on my new computer almost approaching what my 486 could do in 1993. This is not sarcasm: the technical achievement *is* objectively impressive; the route from hand-written C code machine-translated to JavaScript, then after the network transfer loaded onto my client, turned into some intermediate representation and from there to native machine language running in a sandboxed environment, is not trivial by any mean. Considering that by stacking abstraction layers one on top of another their overheads *multiply*, this result is good; and I am sure that there exist problem instances where this composed multiple machine-translation works even move efficiently. Whether such an organization makes sense, of course, is an entirely different matter.

This is not all. I have seen *interpreters* for high-level scripting languages, originally written in C and already quite inefficient when natively compiled, machine-translated into JavaScript to be run over a JIT in a client web browser, in order to interpret script code downloaded from the web. The performance impact is *not* invisible, and the supposed advantage questionable at best: being able to run page scripts in some other (very similar) language rather than JavaScript, or running programs without having an interpreter installed—because in 2021 the voice of the crowd says that people who run software and just *compute on their computers* are no longer hip.

This craziness must stop.

As competent programmers with a social conscience it is our duty to oppose and reverse the dumbing down of software, to empower users again, to make software architectures more rational and, where possible, simpler; I believe that this entails, as a first step, moving applications away from the web and back to local computers directly controlled by the users. Redundant abstraction layers must all be stripped away, mercilessly.
Without the illusion of changing irrational minds with rational arguments we should keep proposing alternative solutions; performance can be one advantage to present, as long as we use VMs *sensibly.* The current popularity of laughably bad practices in the field might make our job easier and unfold into a competitive advantage for our side.

When to use VMs

Even very complex programs run best when they are encoded in the language spoken by the hardware, which is native machine language. Only in favorable cases and with enormous effort some VMs may approach the same level of performance.
Providing some form of JIT compilation is nowadays considered more or less a requirement for a language VM; still, supporting a JIT is necessary but almost certainly not sufficient to rival native performance.

The correct way of employing language VMs is not as an alternative to compilers, but *as an alternative to interpreters.* A few cases come to mind: programs which are subject to modify themselves often at run time, and lightweight utilities where code generation is necessary but complex compiler optimizations are too costly to pay off with respect to the program expected run time—regular expression matchers being an example. A program should employ a VM only for the dynamically generated part of the code, compiling the rest to machine code ahead of time. This yields better performance and allows access to better tools for developing and debugging the bulk of the program.

Adopting a VM as a compiler target instead of a full-fledged assembly backend might be a practical stopgap solution in some circumstances, but it is unrealistic in such cases to expect spectacular results: compiling to a fast VM will yield, in the best case, second-tier performance in the range of good ML or Lisp implementations—Usually simple, sensible systems written in a competent way, but without any concrete ambition of competing in speed with low-level designs.

The details of course will depend on the VM language, and the established practice we are competing against. A VM for an *untyped* system will probably beat even a more sophisticated VM for a dynamically-typed language where every primitive operation requires run-time type checks. Intelligent VM designs will win against naïve versions: if any conditional can be moved from run time to VM code generation the resulting speedup is almost always worth the trouble: translating variable names into register numbers, stack slot indices or static chain distances is a good idea, and simpler systems performing run-time searches on variable names will be easily outperformed. [*FIXME*: **People just don't care, not playing the same game: scripting languages. We can beat even their JITs if we use untyped runtimes.**] [*FIXME*: **is the intent of what I'm saying clear? I want to say that naïf systems will be beaten by less naïf systems, but the opponents' naïveté should not be an excuse for making our systems inefficient. No, I don't think it's clear: I should say this.**]

As stated above in my discussion of decoupling it is important that a VM be specifically designed for the system it supports, to execute its code in the most efficient possible way; for this reason the VM language should ideally not be an API to be published as stable and set in stone, but should rather be subject to change in any future version of the software, to allow for performance tuning. Many details will depend on the language being interpreted, and the performance tradeoff of some choices may not be obvious; for example parameter passing in a VM can employ registers, stacks, or SPARC-style register windows; it is entirely reasonable that a software author may want to switch from one solution to another in a new release, or even as a configuration-time decision based on hardware machine features.

The paragraph above holds as an argument against the reuse of *entire* language virtual machines. Still some of the code involved in VMs, at a different level, can and should be reused.
Even if writing a threaded virtual machine is easy, doing it with attention to efficiency requires some effort. The idea of replication, which I have come to regard as essential, entails in practice automatic code generation techniques which, if conceptually simple, take time to implement. The patch-ins infrastructure, if again intuitive to understand, is laborious to implement and requires very subtle mechanisms (see Section 11.7 [Defective VM instructions], page 42) to guarantee correctness in every case. Several features, technically optional but very beneficial for performance, rely on a small amount of hand-written assembly code. Interestingly such code is specific to the architecture but completely independent from the VM. All these reasons make *the VM construction infrastructure*, rather than VMs, worth sharing.
Jitter is a solution for such reuse.

Software and hardware requirements

[*FIXME*: **Anything non-GNU is secondary; some effort will be made to run with just a standard ANSI C compiler supporting only the simplest dispatch, as long as the supported C standard is recent. There is no support yet, but I'd like a Jittery system to run on the metal, in kernel mode if there are protection rings.**]

[*FIXME*: **What machines are "small". 8-bit machines are too small. 16-bit machines might do, as long as their address space is wider than 16 bits. RAM is the issue. 8MB should be enough for good performance, and less might still be workable; 32MB is plenty. Some very small machines might still have enough RAM to host a Jittery system using threaded code, as long as some modification is made to the runtime not to unconditionally rely on `mmap`; contact me if you want to work in this direction.**]

Design goals

[*FIXME*: **this should be short**]

Performance

[*FIXME*: **Realistically, a Jittery system will not be the fastest possible; a static system compiled with GCC will always win. But I can certainly aim at a different optimum: Jittery programs can be much more expressive. Jittery code will not be faster than native code, but it can be usually within a factor of 2 and almost always within a factor of 5, with Jitter being 1,000 times smaller than GCC, and code generation immensely cheaper.**]

[*FIXME*: **See [The price in performance], page 7**]

[*FIXME*: **See Section 6.5 [Performance tips], page 25**]

Generality

[*FIXME*: **sensible VM designs should be implemented. And even the non-sensible ones I dislike. Experimenting should be easy**]

Minimizing and factoring machine dependencies

Jitter already supports several CPU architectures: see Section 11.9 [Architecture-specific notes], page 42.

For not explicitly supported machines direct-threaded code is always available as a fallback option, which is as portable as GCC; even minimal-threading dispatching might work with no explicit porting effort where `mmap` is available and GCC's `__builtin___clear_cache` suffices to invalidate an icache range along with any necessary dcache flush.

In the unlucky and mostly theoretical case that GCC support were not available for a CPU the user can always revert to `switch` dispatching, which is supposed to rely only on an ANSI C compiler. [*FIXME*: **Can I run at least minimal-threading dispatch with GCC on a non-GNU platform? I think so; I should test on some free software BSD system [Answer: I can, as long as there is an `mmap`. This should include BSDs and exclude Wine, but I guess Cygwin would work—not that I care. [*FIXME*: Make the availability of advanced threading models dependent on `mmap` in `configure.ac`.]]]**

While dispatches differ in performance, they are all functionally identical and controlled by the same API.

Jitter is trivial to port to register-based CPU architectures allowing for a sufficiently wide addressing space: see Section 11.10 [Porting], page 47.

[*FIXME*: **The kind of architecture-dependent code factoring described here has little to do with using VMs for portability: the point here is using VMs for expressivity and having the generated VM run easily on many platforms, as the machine-dependent code is factored within Jitter. If anything, this is about portability *at a meta level*, in the interest of the VM implementor rather than the user. Anyway if people want to use a Jittery VM as a portability layer, against my recommendations, they can.**]

Expressivity and convenience

The ease of use of a VM is a secondary goal compared to its performance, but working on Jitter convinced me that, luckily, a convenient C API and intuitive debugging tools are at the very least *not incompatible* with the need for fast execution on a wide range of CPUs.

Right now a program relying on a Jittery VM has to generate (unspecialized) VM instructions explicitly, in particular taking care or register allocation for register-based VMs. A Jittery VM right now is probably easier to use than a simple JIT requiring the same explicit register handling by the user, but less convenient than JITs which accept high-level expressions. [*FIXME*: **but see Chapter 10 [Writing code generators], page 39**]

[*FIXME*: **Some optional facility for compiling higher-level code involving expressions (with jumps) can be added to Jitter in the future; the fact that the VM details are specified by the user makes an automatic mapping from high-level to low-level code more difficult, but I think these problems can be solved.**]

Non-goals

[*FIXME*: **code size. Particularly residual argument packing. Even if I might play with indirect threading in the future, for fun**]

[*FIXME*: **object orientation: absolutely no, never in one million years. I don't believe in it and I never use it; you can build OO functionalities with a Jittery VM, but OO should not be a fundamental mechanism embedded in all data. Procedures are immensely more important than methods. [*FIXME*: but see Section 10.4.6 [Late binding], page 39]**]

[*FIXME*: **closures, function objects. I don't dislike those, but again they are not fundamental and can be implemented on top of other control primitives. Non-closing procedures are in practice more important than closures.**]

[*FIXME*: **type tagging for all data: absolutely not. That is a recipe for destroying performance. It can be implemented, if needed, within some VM instructions. Doing that systematically would be a mistake. In a future version of Jitter I plan to provide (optional) generation of tagging and untagging C macros, relying on data specification as provided by the user; the exact data representation will be chosen by Jitter at generation time. Of course a user will still be free to do it in her own different way.**]

[*FIXME*: **exceptions: no, not a fundamental mechanism and way too dependent on the execution model (but I'm not against them). They can be implemented.**]

[*FIXME*: **garbage collection: strictly optional (see Chapter 9 [Garbage collection], page 32), and a GC different from the one included in Jitter must be easy to interface to the system. Even one with precise pointer finding.**]

[*FIXME*: **safety features: static code verification, sandboxing: no, these kill expressivity, performance, or both. Static checks can be added to the VM code generator, and dynamic checks to instructions. Personally I'd make them optional, but you're free to make things different. Running code downloaded from who knows where should be discouraged.**]

[*FIXME*: **external representation for code, to be saved and loaded: I am not a fan of the idea and it is not implemented right now; anyway it could be added without a lot of pain; the only real problem is just that some constant expressions involving machine parameters such as the word size would need to be stored symbolically, in order to be evaluated at load time. This can be done quite easily in the existing VM frontend; the C API would get uglier, but maybe not too much. The thing can be done in a VM-independent way, or possibly in a VM-dependent way yielding a very compact binary format. Another problem, in the future, will be data embedded in routines** [*FIXME*: **did I mean what are now called snippets?**]**. Since introducing tagging in every word is out of the question, data would be in a state where saving is possible only right after initialisation. Again, this may complicate the C API.**]

Comparison with other systems

The price in performance

- Register availability [*FIXME*: **write**]
- Calling conventions [*FIXME*: **write**]
- Flags [*FIXME*: **write: special-purpose flag registers are not natural to model explicitly in C, and therefore in Jitter. In particular it is not practical to keep them in a known state across VM instructions. Architectures with no special flag registers (RISC-V, MIPS, Alpha) work in a natural way with no penalty; the others can be made to behave like them, at some performance loss.**]
- Machine idioms [*FIXME*: **write; delay slots, VLIW. The SH case, which may be an extreme unsolvable case**]
- Instruction scheduling [*FIXME*: **write**]
- Memory usage [*FIXME*: **write; modest impact. Portability lost only with respect to very small machines, 16-bit and below: a few megabytes are enough for running a Jittery VM.**]

[*FIXME*: **A compromise involving some loss of low-level control, in exchange for portability and flexibility.**]

Comparison with ahead-of-time compilers

[*FIXME*: **something**]

[*FIXME*: **No very sophisticated compiler to reuse; vectorization is a good example**]

Comparison with JITs

Even as it is, Jitter's frontend should run much faster than the alternative of spawning `gcc` or even just `as` plus `ld` processes, and then using `dlopen` to link object code.

[*FIXME*: **JITs: Jitter is more portable and I believe it can generate code of similar quality. Jitter is not yet as easy to use as JITs working with high-level expressions and abstracting over register allocation: right now some optimization effort has to come from user code (see [Expressivity and convenience], page 6). One could argue that an in-memory assembler is a form of JIT; compared to that alternative a Jittery VM is *way* more convenient to use, but the combination of rewriting and specialization will probably always be slower than in-process assembling. I believe that this last point will not cause a bottleneck in the entire application, but this is a conjecture to be validated experimentally.**]

C is only an implementation tool

There is no dependency on the C runtime model: for example switching the current call stack (when any exists) can be done from a VM instruction, with or without copying, to implement the equivalent of continuations or just cooperative threads or coroutines. A Jittery system can support a garbage collector with exact pointer finding and a custom memory allocator (see Chapter 9 [Garbage collection], page 32), if so desired; runtime information about the type of each object can be made available at runtime if so desired—or not, when not needed. Jitter makes it easy to experiment with unusual runtime data structures, for example a Forth-style dual-stack architecture, the stack-plus-accumulator organization adopted by Caml bytecode, or Appel's "Compiling with Continuations" model [Appel 1992] relying on a fixed number of registers and *no stack*, with function activation records allocated on a garbage-collected heap. Even C ABIs are only relevant when a Jittery VM calls external functions written in C.

Jittery VMs' reliance on C at compile time does not prevent them from deviating from C's runtime model, possibly to a radical extent. In this sense a Jittery VM is not less general than a JIT directly emitting native machine instruction under user control.

Comparison with Vmgen and GForth

[*FIXME*: **vmgen**]

[*FIXME*: **I've learned a lot from the GForth people, particularly Anton Ertl's publications.**] [*FIXME*: **What's new in Jitter: specialization, including instruction rewriting; and I'm particularly proud of my `JITTER_BRANCH_FAST` hack (see [Fast labels], page 42). I'm unconvinced about Ertl's [et al] proposal about literal patching, particularly with respect to GOTs, so I've solved the problem differently. My approach requires more manual porting work, despite remaining trivial (see Section 11.10 [Porting], page 47), but in practice I believe that porting will be possible to a wider range of architectures—essentially, *all* register machines with a sufficiently wide addressing space. With the crucial exception of fast labels my instruction immediates are slightly less efficient than in Ertl's proposal. In most instances this disadvantage is canceled, if not reversed, by specialization on immediates.**]

When using advanced dispatches [*FIXME*: **Jitter**] is more strongly reliant on GCC, and therefore more fragile, than Vmgen—and similar in this respect to `gforth-fast`.

[*FIXME*: **Understandably, GForth and Vmgen are optimized for stacks and nullary VM instructions; Jitter is an attempt to generate efficient VM code using registers and VM instructions with arguments. Jittery VMs are also efficient with nullary VM instructions, which are just a particularly easy subset of the cases to handle, and with Forth-style stacks, which are simple to implement with two registers each when using the TOS optimization, and just one each when not.**]

[*FIXME*: **My computers are named after the computer scientists I admire. I've named my most powerful machine '`moore`' after Chuck Moore, but I suspect Mr. Moore wouldn't name**

his computer after me. Jitter violates his ideas about not being overly general until the need arises.]

License

I believe free software projects have been using permissive licenses too much at least in the last ten years, in practice providing tools to the public on which to develop proprietary software to the disadvantage of the community. I want to reverse that trend. Everybody is welcome to use Jitter, but it and the machines it produces will be covered by a strong copyleft. If there has to be a competitive advantage, let that be in favor of free software.

Jitter has been accepted as part of the GNU Project in late 2021.

I may assign Jitter's copyright to the Free Software Foundation, which will then have the power to decide on the specific license as long as Jitter remains free software. If RMS decided to use a more lax license for some strategic reason I would trust his judgement; indeed I cannot think of anybody I would trust more on these issues. But as long as I legally control this software, its code will be distributed only under strong copyleft.

This is free software, released under the GNU General Public License, version 3 or later.

Contacting the author

You can find my contact information on my personal web site `http://ageinghacker.net`.

I enjoy discussing technical issues in a public forum; there is no mailing list specifically about Jitter right now, but you are free to use the epsilon public mailing list `epsilon-devel@gnu.org` for talking about Jitter as well. Still, I particularly value and respect privacy: if for any reason you prefer to communicate with me in private you can also contact me at `positron@gnu.org`.

I might add a new mailing list specific to GNU Jitter soon.

As long as you respect the license you are welcomed to use Jitter in any way you like, even if your views about language VMs, or indeed about anything, happen to differ from mine (see [About language virtual machines], page 2). If you are building something fun or ambitious with Jitter I will appreciate a message from you, in case you feel like tickling my ego. Otherwise do not worry too much: my ego will survive by metabolizing its own fat for a very long time.

Happy hacking.

Part I
Tutorial documentation

1 Working with the Jitter sources

This chapter explains how to build GNU Jitter from its source code, recommending a specific directory organization which will also be adopted in the following. It does not cover package systems such as the one from the Debian GNU/Linux distribution. As of early 2021 Jitter has not yet been packaged by any distribution; anyway building it from the sources is trivial, and has the important advantage of providing access to the test suite.

Jitter follows the standard GNU build conventions and relies on Autoconf and Automake (see *The GNU Autoconf Manual* and *The GNU Automake Manual*). It also uses Gnulib (see *The Gnulib Manual*) (*only* for portability of the code-generating utility and examples), and GNU Libtool (see *The GNU Libtool Manual*) for handling shared libraries.

Jitter release tarballs have no mandatory compile-time dependencies other than what is normally needed to build programs written in C: a C compiler, C library headers, `make`, and a few standard utilities found in any Unix system such as `sed`. Jitter is supposed to be very portable, but is developed and routinely tested only on GNU systems. Advanced VM dispatching modes require GCC (see *The GNU Compiler Collection Manual*).

The way of building Jitter will look familiar and unsurprising to anyone accustomed to the GNU build system, the only exception being the extra provision for running the test suite on cross-compiled programs through an emulator.

Users unfamiliar with the GNU build system should start from the generic `INSTALL` file.

Jitter development snapshots downloaded from the git repository also contain `README-hacking`, an additional text file with information for users working with unreleased versions. Working from a git snapshot requires git, of course, plus a few tools hopefully already installed on developer machines such as GNU Autoconf, GNU Automake, GNU Libtool, Flex, GNU Bison, GNU Texinfo and GNU help2man. `README-hacking` contains the whole dependency list, and in any case the `configure` script will check for each requirement and complain if any is missing.

1.1 Building conventions and tips

[*FIXME*: **directory organization, cross-compiling directories, reversible installation, stow, what a good build machine is, this is for GNU and likely BSD and everything else is untested**]

1.1.1 Cross-compiling [probably not: better to merge this into the previous sections]

[*FIXME*: **(see *The GNU Stow Manual*)**]

[*FIXME*: **Cross-compilation: why cross-compiling Jitter is useful but a “cross-Jitter” doesn’t make sense**]

1.2 Configuration

[*FIXME*: **be sure to read the output from `configure`; if `configure` fails without a clear error message, showing a syntax error with a line number, please send a bug report including information about your system, your `configure` command line and the complete output.**]

1.3 Generating the documentation

[*FIXME*: **new versions of this manual; optional; the Info version is already there in tarballs**]

1.4 Building

After configuration has succeeded building Jitter should be a simple matter of running '`make`'. This will generate the `jitter` program in the `bin` build subdirectory, plus one runtime library per enabled dispatch. Building should be fast.

At this point it is recommended to build the examples as well, by typing '`make examples`'. Building examples entails executing the `bin/jitter` program generated earlier, which is not possible under cross-compiling (as `bin/jitter` in this case will match the *host*, rather than the *build*, machine) unless an emulator was specified; if using an emulator, this is a good way of testing it for the first time on a relatively simple program. Building the examples involves compiling large generated C files, which will be slower than building `jitter` and the libraries, and will also require more RAM.
If everything works the example programs will appear in the `bin` build subdirectory along with `jitter`. Again the example programs will be one per dispatch, each with the name `uninspired--` followed by the dispatch name. The "Uninspired" VM, if not terribly original, is a representative example of a Jitter virtual machine, to be discussed later in some detail: see Chapter 2 [Tutorial], page 16.

It is still possible to cross-compile the example programs even without an emulator, by copying the generated C files from another build—for example a native build on the same machine. After copying `example-vms/uninspired/*.[ch]` from another *build* directory (the Uninspired generated files are not sources, and are not distributed either) to `example-vms/uninspired/` under the current (cross) *build* directory '`make examples`' should succeed; at that point the generated files `bin/uninspired--*` can be moved to the host machine and run.
If the generated C files copied from elsewhere are not used by '`make examples`' the reason may be a different timestamp: as a simple workaround you may `touch` them: the generated files have to be more recent than their dependencies for `make` not to attempt to regenerate them.

The test suite will run the "Uninspired" VM programs under each enabled dispatch, building the Uninspired VM executables as dependencies if not already present.

[*FIXME*: **Parallel `make` runs with '`-j`' are supported. Having a build directory separate from the source directory is supported, and in fact encouraged.** [*FIXME*: **I suspect that does not work with non-GNU make: I can check**] `ccache` **is supported.**]

1.5 Running the test suite

The test suite works by compiling the Uninspired VM using all the enabled dispatching modes and then running small routines on top of it, comparing the program output against expected results. The test suite is supported in native configurations, and even on cross configurations as long as a working emulator is available to run the compiled VMs.

You can execute the test suite by simply typing `make check`.

The test suite will display a detailed textual report mentioning each test case, with color on most configurations—any red text indicates a problem. The test suite output closes with a summary such as:

```
============================================================================
Testsuite summary for Jitter 0.9.0.607-1076-dirty
============================================================================
# TOTAL: 636
# PASS:  636
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
============================================================================
```

If the number of failures is zero, like in this example, then the test suite detected no problem; otherwise the output above the summary needs to be checked more carefully.

The test suite relies on the functionality in GNU Automake and uses the TAP testing protocol; even used in a simple way as it is TAP allows for a fine-grained view about the specific nature of the failure, which helps when investigating a problem. GNU Autoconf is also used for shell portability, and to check for optional utilities to be used at test time when available. Most of the functionality used across multiple tests is factored in `tests/utility.in`; each subdirectory of `tests` contains one *test*, as a shell script (to be preprocessed at configure time in the source tree, and already preprocessed in the build tree), plus associated data files. For example the source tree file `tests/interpreter/interpreter.test.in` is a shell script for the '`interpreter`' test, to be preprocessed into the build tree file `tests/interpreter/interpreter.test`. Each test may contain many individual *test cases*, each with a 1-based index and a short human-readable name. Each test case may pass, fail, or *fail expectedly* when a feature is known to be broken. Some test cases will be skipped, for example when a dispatching mode is not enabled, or if the user doesn't have Valgrind available to check for memory leaks, of again if the test case is only defined on configurations different from the current one, either lacking or not requiring a specific feature.

Any '`FAIL`' line is a symptom of a problem and should never be ignored. An '`ERROR`' line indicates an unexpected problem in the test script itself, and counts as a bug to report; please do so.

The numbering of test cases will change across Jitter versions, but some reasonable care will be taken to keep test case names stable; if you are reporting an unexpected failure, along with the usual information, please quote the exact line in question, for example

```
FAIL: tests/interpreter/interpreter.test 44 - count-down minimal-threading
```

The test suite subdirectory in the build tree will also contain data files potentially useful to analyze the problem. Unless such files are very large please include them in your reports as well. In the case above you may include information about the failed '`interpreter`' test case 44 named '`count-down minimal-threading`' by sending a tarball containing the relevant files. You can easily generate such a file from the build directory:

```
tar c tests/interpreter/interpreter-44.* | gzip -9 > interpreter-44.tar.gz
```

[*FIXME*: **be sure to check particularly in critical cases: unusual platform, new GCC version, new Jitter version**]

[*FIXME*: **in case some test fails reconfigure disabling the offending dispatch and try again**]

[*FIXME*: **A test case hanging is bad. The Uninspired driver as invoked from the test suite is supposed to fail after a timeout and the test scripts contain similar provisions, but the machinery is not bulletproof, particularly with incorrect machine code running under emulators or on non-GNU systems.**]

1.5.1 A test suite is not a correctness proof

[*FIXME*: **even all passes is not a correctness proof. QEmu, by design, doesn't emulate L1i invalidation**]

1.6 Installing

2 Tutorial

2.1 Using a VM

2.1.1 Predefined frontend

2.1.2 Disassembly

2.1.3 C API tutorial

2.1.4 Building tutorial

2.2 A VM definition

Part II
Reference documentation

3 Invoking `jitter`

4 Standalone VMs

5 Compiling Jittery programs

5.1 Generating and compiling C code

5.1.1 Cross-compiling VMs

5.2 Building a Jittery program

5.2.1 Building preliminaries

[*FIXME*: `vmprefix-vm2.c` is fragile to compile, and may require its own specific *JITTER_CFLAGS* for correctness. The rest is actually not fragile, but you still need to pre-process every file using any Jitter header with *JITTER_CPPFLAGS*; this restriction helps preventing conflicts between a sub-package Jitter and a different installed version, and makes Jittery VMs behave consistently with Jitter as a dependency or as a sub-package.]

[*FIXME*: *JITTER_CPPFLAGS JITTER_CFLAGS JITTER_LDFLAGS JITTER_LDADD*. There are also dispatch-specific versions of these, but most people will only use the fastest dispatch available on their configuration. Any C file including any header from Jitter must be preprocessed with the flags in *JITTER_CPPFLAGS*.]

5.2.2 Invoking `jitter-config`

5.2.3 Autoconf and Automake

[*FIXME*: the files `configure.ac` and `Makefile.am` in the Jitter distribution are much more complex than the ones needed to build an ordinary Jittery program, because they have to deal with a non-installed `jitter` and exactly specify dependencies on files which no longer change after installation. Jitter's own build system also makes a considerable effort to support cross-compilation in a convenient way including a running a cross-`jitter` and even the test suite through an emulator, in an effort to test the critical part of the software as thoroughly as possible. Ordinary Jittery programs will avoid going to such lengths. The recommended way of using Jitter right now is *sub-package mode*: see Section 5.2.3.2 [Jitter as a sub-package], page 21.]

5.2.3.1 Jitter as a dependency

5.2.3.2 Jitter as a sub-package

5.2.4 Make

5.2.5 Other systems

5.3 Non-production Jittery programs

5.3.1 Debugging Jittery programs

5.3.2 Profiling Jittery programs

[*FIXME*: Explain the idea.]

[*FIXME*: Preprocess Jitter-generated files (*`vmprefix`*`-vm1.c` and *`vmprefix`*`-vm2.c`) with at least one of these macros defined: *`VMPREFIX`*`_PROFILE_SAMPLE` *`VMPREFIX`*`_PROFILE_COUNT`]

[*FIXME*: Explain the implementation limits and give details about the overhead. Sample profiling is more accurate with complex dispatches, where it is possible to guarantee that sam-

pling takes place at the same time during execution of each VM instruction — the beginning. The current profiling support is non-reentrant and does not play well with multithreading. Profiling is unreliable under emulation, particularly with sophisticated JIT-type emulator like QEmu.]

[*FIXME*: **Example: profile `nop` on Uninspired.**]

No-threading, x86_64:

```
# nop (19 B):
    0x00007f15f0d7f244 48 c7 43 e8 62 01 00 00  movq   $0x162,-0x18(%rbx)
    0x00007f15f0d7f24c 48 8b 43 e0              movq   -0x20(%rbx),%rax
    0x00007f15f0d7f250 48 ff 80 10 0b 00 00     incq   0xb10(%rax)
```

No-threading in a less favourable case, 32-bit MIPS.

```
# nop (40 B):
    0x3fd7f290 24030162        addiu   $3,$0,354
    0x3fd7f294 ae03fff4        sw      $3,-12($16)
    0x3fd7f298 8e03fff0        lw      $3,-16($16)
    0x3fd7f29c 8c620b14        lw      $2,2836($3)
    0x3fd7f2a0 8c650b10        lw      $5,2832($3)
    0x3fd7f2a4 24440001        addiu   $4,$2,1
    0x3fd7f2a8 0082102b        sltu    $2,$4,$2
    0x3fd7f2ac 00451021        addu    $2,$2,$5
    0x3fd7f2b0 ac620b10        sw      $2,2832($3)
    0x3fd7f2b4 ac640b14        sw      $4,2836($3)
```

[*FIXME*: **Driver:** `--profile-unspecialized` `--profile-specialized`]

6 Writing VM specifications

6.1 Syntax

6.1.1 Header

6.1.2 Embedding C code

6.1.3 Meta-instructions

6.1.3.1 Meta-instruction headers

6.1.3.2 Meta-instruction attributes

`non-relocatable`
`cold`

`branching`
: [*FIXME*: **The following macros are only available in branching macros: `JITTER_EXIT`, every macro with `_BRANCH_` in its name**]

`caller`

`callee`

`returning`

Using a Jitter macro not available from some instruction lacking the required attribute yields a compile-time error message, which will be friendlier and more explicit when compiling with GCC.

[*FIXME*: **Talk about `JITTER_LINK` and branch-and-link branches.**]

6.1.3.3 Embedding C code in meta-instructions

6.1.3.4 C macros available to VM instructions

`JITTER_SPECIALIZED_INSTRUCTION_OPCODE`
`JITTER_SPECIALIZED_INSTRUCTION_NAME`
: [*FIXME*: **This may be a bad idea. It is very difficult to use correctly with replicated code, out of non-relocatable instructions. A good use of this is for comments in inline assembly.**]

`JITTER_SPECIALIZED_INSTRUCTION_MANGLED_NAME`

`JITTER_ARG`*i*
`JITTER_ARGP`*i*
`JITTER_ARGU`*i*
`JITTER_ARGN`*i*
`JITTER_ARGF`*i*

`JITTER_FAST_REGISTER(class, index)`
`JITTER_FAST_REGISTER_`*c*`_`*i*

/* Expand to the i-th register, which must be a slow register, as an lvalue. The given index must be a register index counting from 0 and including fast regusters as well, if there are any. For example if an r class had 3 fast registers then the first slow register would be %r3, to be accessed as `JITTER_SLOW_REGISTER(r, 3)`. It would be invalid to access %r0, %r1 and %r2 which this macro, as %r0, %r1 and %r2 would be fast. */

`JITTER_SLOW_REGISTER_`*`c_i`*

/* It's not possible to have a single macro JITTER_REGISTER taking an index and expanding to either a fast or a slow register lvalue, due to CPP conditional limitations. This restriction is unfortunate, but we have to live with it as long as we don't switch to a different preprocessor. What we can have is a set of zero-argument macros each expanding to a register lvalue, for *a limited number* of registers. Here we define access macros for every fast register plus a reasonable number (currently 32) of slow registers, per class. */

`JITTER_REGISTER_`*`c_i`*

for every class *c* and for enough indices *i* to cover every fast register and 32 slow registers.

`JITTER_BRANCH(`*`where`*`)`
`JITTER_BRANCH_FAST(`*`where`*`)`

`JITTER_BRANCH_AND_LINK(`*`where`*`)`
`JITTER_BRANCH_FAST_AND_LINK(`*`where`*`)`

`JITTER_LINK`

`JITTER_BRANCH_FAST_IF_ZERO(`*`e0, where`*`)`
`JITTER_BRANCH_FAST_IF_NONZERO(`*`e0, where`*`)`
`JITTER_BRANCH_FAST_IF_POSITIVE(`*`e0, where`*`)`
`JITTER_BRANCH_FAST_IF_NONPOSITIVE(`*`e0, where`*`)`
`JITTER_BRANCH_FAST_IF_NEGATIVE(`*`e0, where`*`)`
`JITTER_BRANCH_FAST_IF_NONNEGATIVE(`*`e0, where`*`)`

[*FIXME*: **Document or possibly even remove the non-fast variants of these.**]

`JITTER_BRANCH_FAST_IF_EQUAL(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_NOTEQUAL(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_LESS_SIGNED(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_NOTLESS_SIGNED(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_LESS_UNSIGNED(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_NOTLESS_UNSIGNED(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_GREATER_SIGNED(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_NOTGREATER_SIGNED(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_GREATER_UNSIGNED(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_NOTGREATER_UNSIGNED(`*`e0, e1, where`*`)`

[*FIXME*: **Document or possibly even remove the non-fast variants of these.**]

`JITTER_BRANCH_FAST_IF_AND(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_NOTAND(`*`e0, e1, where`*`)`

[*FIXME*: **Document or possibly even remove the non-fast variants of these.**] [*FIXME*: **Explain how not-and is different from nand, and why we do not want nand here.**]

`JITTER_BRANCH_FAST_IF_NEGATE_OVERFLOWS(`*`e0 where`*`)`
`JITTER_BRANCH_FAST_IF_PLUS_OVERFLOWS(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_MINUS_OVERFLOWS(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_TIMES_OVERFLOWS(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_DIVIDED_OVERFLOWS(`*`e0, e1, where`*`)`
`JITTER_BRANCH_FAST_IF_REMAINDER_OVERFLOWS(`*`e0, e1, where`*`)`

[*FIXME*: **Document or possibly even remove the non-fast variants of these.**]

```
JITTER_NEGATE_BRANCH_FAST_IF_OVERFLOW(et, e0, where)
JITTER_PLUS_BRANCH_FAST_IF_OVERFLOW(et, e0, e1, where)
JITTER_MINUS_BRANCH_FAST_IF_OVERFLOW(et, e0, e1, where)
JITTER_TIMES_BRANCH_FAST_IF_OVERFLOW(et, e0, e1, where)
JITTER_DIVIDED_BRANCH_FAST_IF_OVERFLOW(et, e0, e1, where)
JITTER_REMAINDER_BRANCH_FAST_IF_OVERFLOW(et, e0, e1, where)
```

[*FIXME*: **Document or possibly even remove the non-fast variants of these.**]

[*FIXME*: **document stacks operations**]

6.2 VM states

[*FIXME*: **Blocks:** `state-initialization-c` `state-finalization-c` `state-reset-c`]

[*FIXME*: **Access macros evaluating to lvalues, with either one (from VM instructions) or two arguments (elsewhere):** `JITTERLISPVM_STATE_RUNTIME_FIELD` `JITTERLISPVM_STATE_BACKING_FIELD`]

[*FIXME*: **Make these macros usable, with one argument (or, better, with two: in that case specify also the formal name) from the code blocks mentioned above.**]

6.3 Instruction rewriting

6.4 Emacs major mode for Jitter

6.5 Performance tips

[*FIXME*: **for register machines having a lot of registers is important.** [*FIXME*: **is it possible to have too many? I see performance degradation on x86_64, but that is probably due to bad choices in residual registers.**] **Keeping the number of meta-instructions smaller, and making the non-critical ones non-relocatable, can buy you more fast registers.**]

[*FIXME*: **number of instruction arguments: residuals have a cost at run time; non-residuals have a cost at compile time, and may make the VM too large to compile**]

[*FIXME*: **optimize for the common case: for example making division non-relocatable will probably not be a problem: the time spent in division will hardly ever be significant, in almost any routine. This is probably true for multiplication as well, unless you're doing graphics. You can always optimize the common (and cheaper) cases of multiplying by, dividing and taking the reminder or a power of two with rewrites into different, relocatable instructions:** $a\ 2^b$ **can be rewritten into '**`a << b`**',** $a/2^b$ **into '**`a >> b`**' (as long as** a **is unsigned) and** $a\ mod\ 2^b$ **into '**`a & ((1 << b) - 1)`**'. The rewrite targets can have one specialized literal.**]

[*FIXME*: **CISC vs RISC: cite the APL argument** [*FIXME*: **my previous deleted text, to reword:** *The opponents of this argument point out at languages with very complex primitives such as APL, where the interpretation overhead becomes proportionally less significant compared to the time spent in primitives, each one performing a greater amount of useful work than in other designs. The counter-argument has the merit of recognizing one source of the problem, and that languages like APL are indeed easier to deal with than others. Still, the part of the community interested in playing with the fundamental elements of computation, which certainly includes me, will also demand systems providing control on a finer granularity, where each fundamental operation is kept as simple as possible and more complex behavior only emerges as composition of fundamental operations. This is the case of epsilon, Forth, and to a slightly lesser extent, Lisp.*]]

[*FIXME*: **CISC vs RISC, flags: orthogonality is not mandatory. Having multiple flag registers like on the PowerPC, or using ordinary registers to store boolean conditions like on**

RISC-V, MIPS and Alpha looks neat, but VM flags registers will not be mapped to hardware flag registers: the "pretty" solution on the VM would force three-argument comparison instructions, bloating the instruction space, and on most machines would cost two comparisons instead of one per jump at run time. The efficient solution for a VM in this case is a MIPS-style conditional branch instruction taking two arguments (but on a VM they can be immediate), and a fast label. Flags should be considered inherently transient, and keeping their state in a register across VM instruction will cost hardware instructions. You should resist the temptation of having both quotient and reminder computed by one VM instruction, unless the instruction is non-relocatable.]

[*FIXME*: **On a VM implied operands (for example a frame pointer or even a heap allocation pointer held in a fixed register) are fine, as they reduce the number of VM instruction arguments, limiting their combinatorial explosion and affording you more fast registers. It's perfectly acceptable to design your VM around your ABI: if you change your mind later, you can change the VM easily: it's not silicon.**]

[*FIXME*: **constants, particularly if big: specialize if known in advance.**]

[*FIXME*: **small but performance-critical constants, such as type or garbage-collection tags, or unique objects (`nil` in Lisp): specialize, and define a checking instruction doing only that, so as to keep the other instructions smaller. It is probably better to keep the non-critical path used to handle errors slow, for example keeping the error handler address on a stack, rather than passing slow labels around. Residual arguments are expensive, and should not be passed every time just to handle rare cases.**]

[*FIXME*: **fast branches: use the specific macros for conditional branches, rather than if (condition) BRANCH_FAST(JITTER_ARGF0); The specific macro generates faster code.** [*FIXME*: **show the two disassemblies**]]

6.5.1 Preventing VM instruction defects

[*FIXME*: **make extremely clear that all of this affects performance but not correctness: having defective instructions will not make programs crash (after I fix "the bug", of course); but it will be fixed when people actually read this**]

[*FIXME*: **See Section 11.7 [Defective VM instructions], page 42.** [*FIXME*: **write this: introduction to the problem, examples**]]

[*FIXME*: **decouple control from state updates, examples**] [*FIXME*: **unconditionally updating the VM state and then having a conditional branch within the same VM instruction depending on the new, updated state, is less dangerous and will usually not trigger defects. But I will have to make experiments to confirm this.**]

6.5.2 Performance tips for VM routines

[*FIXME*: **don't waste registers: lower-numbered is better**] [*FIXME*: **VM registers in memory are still better than slow registers.**]

[*FIXME*: **constants, particularly if big: count down rather than up**]

7 C API

7.1 Initialisation

7.2 Finalisation

[*FIXME*: **This has nothing to do with** *garbage collection* **finalisation: see [Defining finalisable shapes], page 34**]

7.3 VM state

7.4 VM routines

7.4.1 Routine structure

7.4.2 Adding labels

7.4.3 Adding instructions

7.4.4 Executable routines

7.4.5 Code execution

7.4.6 Multiple routines

7.5 Running VM routines

7.6 Concurrency

[*FIXME*: **The Jittery C API is thread-safe, including the Flex/Bison frontend. After initialisation no global variables are written, and the code is reentrant.** [*FIXME*: **This is currently true becasue I use a separate call to `mmap` for every native code allocation, but it will get more complicated: in the future I will need a mutex for code allocation, almost certainly hidden in the implementation. I should mention here the dependency on POSIX threads, or as an alternative the need for the user to manually synchronize the one critical operation.**]]

[*FIXME*: **The VM** *state* **data structures encodes the execution state of a single thread. Having different threads working concurrently on different states, possibly executing the same code,** *is* **supported. There is no global interpreter lock.**]

[*FIXME*: **The user needs to take care of synchronization herself, when needed. Of course it is possible to break reentrancy from user code, but it should be easy to avoid that.**]

8 Data representation and tagging

Is it not entirely clear which of the techniques were already known to one of us and which ones we reinvented

From the acknowledgments in [Gudeman 1993].

I share that sentiment.

8.1 Design issues

8.1.1 Boxed and unboxed data

8.1.2 Bit twiddling and tags

[*FIXME*: **heap pointers are aligned: low bits are zero**]

[*FIXME*: **some high bits are also often zero, particularly on 64-bit machines**]

[*FIXME*: **low bits are better than high bits: immediate representation, compatibility with conservative pointer finding, efficiency in memory access**]

[*FIXME*: **shifting: you want it for fixnums, not for pointers**]

[*FIXME*: **sign extension and right-shifting in C: there is a Jitter macro**]

[*FIXME*: **untagging: and versus subtraction**]

[*FIXME*: **should the tagged pointer point to a header rather than to the beginning of user data? On almost all machines in makes no difference; SH the one exception I know.**]

8.1.3 Object headers

[*FIXME*: **It is also possible to have a header which is only useful at collection time, in order to distinguish the object shape and know which fields need to be updated and which must not be touched; still the shape may be distinguishable from its unique pointer tag alone, for efficiency's sake. I plan to do this for JitterLisp closures, where performance is critical: closures are used all the time and the type check at the beginning of a call must be fast.**]

[*FIXME*: *type code* **(the first header word), name from Gudeman. The type code must suffice to recognize the shape of an object, but it does not need to be limited to one constant word per shape: it can contain other information as well.**]

[*FIXME*: **rename "type code" to "shape code"? If I do I have to do it in the sources as well.**]

[*FIXME*: **The remark at [Unique data], page 29, is important: it is the user's responsibility to ensure that no tagged object is ever created which is also valid as a type code**]

[*FIXME*: **Another possible application of unique objects is for inserting as padding in headerless objects where not every word is used.** [*FIXME*: **add a macro to make this easy**]**. It would make perfect sense to have a one-word `#<padding>` object, to be found when scanning the heap.** [*FIXME*: **Document `JITTER_GC_PAD_HEADERLESS_OBJECT`. I could even provide a helper macro `JITTER_GC_ALLOCATE_AND_PAD_HEADERLESS`, but having a very friendly C API might not be that useful: such a macro would not be usable from VM code where heap allocation follows a different pattern.**]]

8.1.4 Shapes are not types

8.1.5 Parsable heaps

[*FIXME*: **Give the definition. We do not require parsable heaps, but a weaker condition which allows for headerless boxed objects. Headerless boxed objects must be made of individually valid tagged objects only.**]

8.1.6 Checking and dispatching

[*FIXME*: **a `type` procedure**]

8.1.7 Tagging recommendations

8.1.7.1 Unique data

[*FIXME*: **Booleans, empty list, EOF, “nothing”, “undefined” values reserved for errors. A “padding” value (see [Object headers], page 28)**]

[*FIXME*: **If variable state needs to be associated to some unique objects the state can be kept in a global table out of the heap, without representing unique objects as pointers; they could be rather tagged array indices. This strategy not involving pointers makes adding new unique values easy.**]

[*FIXME*: **A trick to keep the heap parsable: representing type codes as unique data. Only a finite number is needed, and when they do not contain other fields they are all known in advance. In this case it becomes invalid to make unboxed values with the value of a type code as user objects, in the presence of headerless boxed objects: it would confuse the collector, which would become unable to distinguish an unboxed object from the beginning of a boxed object. Such an error is detected at run time in some cases, only when `JITTER_GC_DEBUG` defined.**]

[*FIXME*: **unique values are usually unboxed, but they can also encode pointers to out-of-heap data — in which case they can be treated as unboxed by the garbage collector.**]

[*FIXME*: **enumerates**]

8.1.7.2 Characters or code points

[*FIXME*: **explain the trick I use in JitterLisp**]

8.1.7.3 Fixnums

[*FIXME*: **a tag of all `0`s is best for fast arithmetic.**]

8.1.7.4 Flonums

[*FIXME*: **a tag of all `1`s is best, when a tag of all `0`s is used for fixnums.**]

[*FIXME*: **`jitter_float` may be acceptable. If not, things are more complicated. Some platforms require double values to be aligned to 64 bits, and others, while not having a strict requirement, suffer from a performance hit from misaligned accesses. `double` values on 32-bit platforms requiring 64-bit alignment will be messy** [*FIXME*: [*FIXME*: **Now I have an excellent solution**]**I have no good solution for this yet. Even adding a padding field to every object would not work for data different from `double` requiring wider alignment (for example vector data). Maybe creating a padding object. I have some experimental code in `~/repos/jitter/my-tests/alignment-and-object-headers/alignment-and-object-headers.c`, not in the git repository, containing the right formulas which are not completely trivial. The best solution is almost certainly raising the minimum heap object size to the maximum alignment, and then pad in a fixed way. Of course heap pages must also be aligned.**]]

Some users might be tempted to save one instruction when untagging by not clearing a non-zero tag after converting a tagged object to floating point, and keep the tag bits as they are as part of the floating-point value; after all a bit pattern of all zeros in untagged floating-point data appears just as arbitrary as the original tag. This intuition has the unfortunate flaw of not allowing for an exact representation of most integers as floating-point data, including zero. Floating-point calculations such as ‘`2.0 * 3.0`’, ‘`2.0 * 1.0`’ or ‘`10.0 * 0.0`’ yielding inexact results and violating algebraic cancellation laws feel hardly suitable to any application.

However the optimization attempt described above succeeds in a different setting, on systems lacking fixnums and where the only available numbers are floating-point. In such a scenario the floating-point tag should in fact be chosen as zero, which eliminates any untagging overhead while keeping small integers representable in floating point with no error.

[*FIXME*: **how inefficient would it be to *round* a fixed-point number at the time of untagging, keeping into account the smaller mantissa? What I am currently doing is efficient but in practice amounts to truncation.** `https://en.wikipedia.org/wiki/Floating-point_arithmetic#Rounding_modes`]

8.1.7.5 Conses

8.1.7.6 Closures_

[*FIXME*: **fast shape-checking is usually important. An additional header with a type code just to keep the heap parsable, however, should not be a problem.**]

8.1.7.7 Primitives

[*FIXME*: **These are heavily used, but we can make them not heavily shape-checked.**]

[*FIXME*: **I recommend the JitterLisp solution: hide primitives from users by providing predefined closure wrappers for them: closure wrappers, while safe when called, internally use primitives in an unsafe way. Compiled VM code or even correctly formed interpreted code based on ASTs does not pay for the indirection, thanks to a simple optimization. This way there will be no need to check for primitive shape at call time**]

[*FIXME*: **Primitives may be tagged indices into a global read-only table, or tagged pointers to non-heap data, or directly to a function. The programmer must guarantee that such out-of-heap data remain aligned somehow, for example with the functionality form `stdalign.h` or the GNU C attribute `aligned`.**]

8.1.7.8 Arrays

[*FIXME*: **they are allocated frequently on a large scale: keep them compact in memory**]

[*FIXME*: **large objects and indirection**]

8.1.7.9 Symbols

[*FIXME*: **since symbols are always boxed and they can be compared by pointer equality, their memory representation does not really matter: in this case headers cost nothing**]

[*FIXME*: **Checking at runtime whether a given object is a symbol (`symbolp` in Common Lisp, `symbol?` in Scheme) is not a common operation in compiled code, which should make a check on the type code acceptable in most circumstances.**]

8.1.7.10 Bignums

8.1.7.11 Possibly large objects

[*FIXME*: **Boxed with a header including one of two possible values: large or non-large, followed by an untagged pointer to the actual data. For non-large data the pointer points inside the same object payload, and it is easily handled by the garbage collector. For large data the pointer points out of the heap. Decoding is straight-line code. Whenever the object is copied the untagged pointer must be updated, which is easy.**].

8.1.7.12 Finalisable objects

8.1.7.13 Code objects

[*FIXME*: **finalisation of some resources different from the garbage-collected heap memory may require reference counting. The C API makes it easy to handle VM routines in this way, with pin and unpin functions already provided.**]

8.1.7.14 Other data types

8.1.7.15 Broken hearts

[*FIXME*: **broken hearts need a type code, not a specific pointer tag: in fact a broken heart will be pointed by a tagged pointer, whose tag will be meaningful; the fact that the object has been forwarded is recorded in memory within the object itself.**]

8.1.8 Custom tagging

[*FIXME*: **Different designs are not as easy to implement as what is recommended here, but at least Jitter should not prevent them. If you find that I am wrong please contact me.**]

[*FIXME*: **BiBOP**]

8.1.9 No tagging

8.2 Tag API

8.2.1 Tag checking in VM code

9 Garbage collection

Jitter includes a garbage collector designed to be compatible with Jittery VMs and suitable to many user systems.

The Jitter garbage collector is optional: according to the application requirements a VM can even operate without any garbage collector at all, or use some other custom memory subsystem. Jittery VMs are also compatible with the Boehm-Demers-Weiser conservative-pointer-finding collector available at `https://www.hboehm.info/gc/`.

`struct jitter_gc_heaplet`

9.1 Garbage collector design

[*FIXME*: **fast allocation, allocating from several threads with no locks, high allocation and low survival rate. Reentrant: no global variables in C exposed to the user. Multiple heaps containing objects of different shapes are possible in the same application at the same time. Portable, no reliance on hardware page faults. Optimised for bandwidth, but usually low lantency as well. No real-time guarantees.**]

[*FIXME*: **portable in practice. Is is difficult not to rely on undefined behaviour when comparing pointers one of which may fall out of allocated space by more than one element (of course I am not speaking of ever dereferencing them, but even computing their value is undefined behaviour in C): since any reasonable implementation in a flat addressing space will allow this and GCC in fact guarantees it** [*FIXME*: **find exact citation**] **I will not be pedantic. I am fine with not supporting 16-bit segmented addressed on i386: in protected mode everything will work as it should.**]

9.1.1 Exact pointer finding and moving

[*FIXME*: **Comparison with Boehm's collector, which is widely used.**]

[*FIXME*: **Some deserved praise for Boehm's work, which is very impressive given the problem he solves. A collector like ours plays the game with different rules.**]

[*FIXME*: **Cite my own attempt: [Saiu 2011]**]

9.1.1.1 API comparison

Let `o1`, `o2` and `o3` be registered roots, and `N` some positive integer. [*FIXME*: `range append`]

[*FIXME*: **Wrong (unregistered roots)**]

```
o1 = append (range (0, N), range (0, N));
print (o1);
```

[*FIXME*: **Correct**]

```
o2 = range (0, N);
o3 = range (0, N);
o1 = append (o2, o3);
print (o1);
```

[*FIXME*: **Also correct, but slightly more subtle**]

```
o2 = range (0, N);
o1 = append (o2, range (0, N));
print (o1);
```

[*FIXME*: **Equally correct**]

```
o2 = range (0, N);
o1 = append (range (0, N), o2);
print (o1);
```

[*FIXME*: **Another subtly incorrect example. The variable** `v` **holds a vector boxed object as a tagged value, and is a root.**]

```
vector_set (v, i, cons_make (EOL, vector_ref (v, i)));
```

[*FIXME*: **The apparently harmless call `vector_ref (v, i)` accesses the vector `v` and returns its `i`-th element, without performing any heap allocation; the problem is that the *result* of the function call is a heap object, which may be moved if `cons_make` performs a collection; if the vector element contained some pointer to an alive object, such as `v` itself, that pointer would become invalid, and a root would be lost. The following variant is correct, with `tmp` being a root:**]

```
tmp = cons_make (EOL, vector_ref (v, i));
vector_set (v, i, tmp);
```

[*FIXME*: **As long as `cons` is defined correctly, which implies temporarily registering both of its arguments as roots, there is no need to use another variable for `vector_ref (v, i)`.**]

9.1.1.2 Performance comparison

[*FIXME*: **A collector performance is only important when the allocation rate is high. An application rarely allocating on the garbage collected heap will only see negligible differences in performance from one garbage collector to another.**]

[*FIXME*: **allocating from VM code can be made extremely fast by inlining fast-path code (see Section 9.3 [Heap allocation from VMs], page 35): here this collector should win by a wide margin.**]

[*FIXME*: **It is possible to design some important boxed shapes to require no headers. This tight representation of objects will improve locality and, along with compaction, possibly mitigate the cache disruption caused by copying.**]

[*FIXME*: **Temporary root registration, while well optimised, still has an impact on performance in** *deeply recursive* **C code where each function needs to register a few roots, call itself, and deregister at the end. Bohem's collector has no similar work to do for registering and deregistering a temporary root, as it just scans the entire C stack at collection time. This problem does not exist in recursive VM code where the stack is scanned via a hook see [Garbage collection hooks], page 34.**]

[*FIXME*: **A moving collector performs well when the survival ratio is low. Mark-sweep is less sensitive to this issue and its performance degrades less drastically as the heap becomes almost full. Generational collection, as implemented in this collector, may attenuate or possibly even reverse this disadvantage — I have not taken measurements yet.**]

[*FIXME*: **From preliminary tests object finalisation (see Section 9.2.2.3 [Defining finalisable shapes], page 34) in Jitter's garbage collector appears to be extremely fast, much better than in Boehm's collector; this will be relevant with some applications but not with others.**]

[*FIXME*: **"Sharing" is slow in Jitter's collector in some circumstances: explain shy**]

9.2 Garbage collector API

9.2.1 Debugging garbage collection

[*FIXME*: `JITTER_GC_DEBUG JITTER_GC_LOG`] [*FIXME*: `JITTER_GC_DEBUG` **checks for problems in user code, as well as for internal collector bugs and problems that may mimic internal collector bugs, by violating invariants. We believe that the very defensive checks enabled in this mode will help when debugging garbage collector problems, notoriously difficult to track. In debugging mode the garbage collector performance will suffer a hit, but not to the point of making the system unusable; in typical cases the speedup should range around** 0.3 **to** 0.7.]

9.2.2 Defining shapes

A user needs to define the shape of heap objects before being able to initialise a garbage-collected heap. See Chapter 8 [Data representation and tagging], page 28, for a description of the design of shapes and practical suggestions.

[*FIXME*: **alignment, headers, type codes and** `struct`**s. Beware of the size of every allocated object, which must be a multiple of** `JITTER_GC_MINIMUM_OBJECT_SIZE_IN_BYTES` **in bytes also counting the header and any internal or final padding. There are macros helping** [*FIXME*: **Follow the definitions for cons and vector in** `jitter-gc-test.c`**, which do the right thing.**]]

9.2.2.1 Defining headerless shapes

9.2.2.2 Defining headered shapes

9.2.2.3 Defining finalisable shapes

[*FIXME*: **gc object finsalisation**]

9.2.3 Garbage collector initialisation and finalisation

[*FIXME*: **This has nothing to do with object finalisation for finalisable shapes: see [Defining finalisable shapes], page 34**]

9.2.4 Root registration

[*FIXME*: **Do not register roots more than once. It leads to disasters, only reliably detected with** `JITTER_GC_DEBUG`**. It is of course possible to register multiple roots, each residing at a different memory address, pointing to the same heap object. Each root must be unique in the sense of** *not residing at the same address* **as any other root.**]

9.2.4.1 Global roots

[*FIXME*: **I should rename these: "global" roots are in fact per-heaplet, even if they have an undetermined life time.**]

9.2.4.2 Temporary roots

9.2.4.3 Garbage collection hooks

[*FIXME*: **pre- and post-collection hooks**]

9.2.5 Heap allocation from C

[*FIXME*: **These should be used when allocating:** `JITTER_GC_HEADERLESS_SIZE_IN_BYTES` `JITTER_GC_HEADERED_SIZE_IN_BYTES` `JITTER_GC_PAD_HEADERLESS_OBJECT` [*FIXME*: **If I end up actually defining this:** `JITTER_GC_ALLOCATE_AND_PAD_HEADERLESS`]]

9.2.6 Write barriers

[*FIXME*: **The write barrier registers the stored addresses in the old generation which may point to young boxed objects.**]

[*FIXME*: **It is possible to avoid the overhead of the write barrier when writing to a boxed object field in two situations:**]

- the object being written into is definitely young
- the object being written is definitely unboxed

[*FIXME*: **In practice the first case only applies at object initialisation time: still, this is an important optimisation:** *initialising writes* **do not need the write barrier, since they are guaranteed to happen between object allocation and the next collection — collecting when some object is still not completely initialised is invalid and leads to disasters.**]

[*FIXME*: **The write barrier** *must not* **be used for updating roots: it is only for updating fields of heap objects. Applying the write barrier to a non-heap objects such as a root is equivalent to registering roots more than once (see [Root registration], page 34).**]

```
/* FIXME: comment well. ///////////////////////////////////////////////////////////////////
```

```
   The result is the new allocation limit.

   Notice that, differently from garbage collecting functions, this function
   gives no guarantee about how much space is freed in the queue; there might
   simply be no place left.  This function will flush the queue and register
   the given pointer, but the next enqueue attempt might fail again.

   Still it should be noticed that, differently from the case of
   jitter_gc_collect_0, this function does in fact guarantee that the requested
   operation succeeds: only *the next* attempt may fail.  This has implications
   over VM code: after jumping to the slow path of the write barrier, control
   should return *past* the attempt: that one write barrier succeeded.
   Compare with allocation from VM code, where the slow path has to jump back
   to the fast path to try again after an allocation failed  [FIXME: is that true for allo-
cation?]. */
jitter_gc_heap_pointer
jitter_gc_write_barrier_queue_flush_1 (struct jitter_gc_heaplet *a,
                                       jitter_gc_heap_pointer allocation_limit,
                                       jitter_gc_tagged_object *pointer)
  __attribute__ ((nonnull (1), nonnull (2), nonnull (3),
                 returns_nonnull, warn_unused_result));
```

9.2.7 Promotion and sharing

9.2.8 Garbage collector tuning

9.2.9 Other garbage collection operations

9.3 Heap allocation from VMs

9.4 GC and threads

9.5 Not yet implemented features

Part III
Performance

10 Writing code generators

10.1 Code generation tips

[*FIXME*: **two stacks**]

[*FIXME*: **compile the uncommon or error path out of line**]

10.2 Stack code generation

10.3 Register code generation

10.4 Specific language features

10.4.1 Primitive operations

10.4.2 Conditionals and Boolean expressions

10.4.3 Procedural abstraction

10.4.4 Dynamic tag checking

10.4.5 Closures

[*FIXME*: **Closures with known code**]

10.4.6 Late binding

[*FIXME*: **Sometimes called dynamic dispatch, but nothing to do with VM dispatches**]

10.5 Calling conventions

[*FIXME*: **Give suggestions**]

10.6 Specific runtime features

10.6.1 Multi-threading

10.6.1.1 Reentrancy issues

10.6.1.2 Iterating on VM states

If this is not documented anywhere explain `vmprefix_states` and the state doubly-linked list.

```
static void
handle_signal (int signal_number)
{
  struct vmprefix_state *s;
  for (s = vmprefix_states->first; s != NULL; s = s->links.next)
    {
      VMPREFIX_STATE_AND_SIGNAL_TO_PENDING_SIGNAL_NOTIFICATION (s, signal_number) = true;
      VMPREFIX_STATE_TO_PENDING_NOTIFICATIONS (s) = true;
    }
}
```

Notice that `vmprefix_states` is also safe to use from VM instructions: it is automatically wrapped as a global.

A pointer to the current state is accessible from VM instruction code as `VMPREFIX_OWN_STATE`. It can be used for pointer comparisons when iterating over all states, and can be useful for a thread notifying every state except its own.

```
#if defined (JITTER_HAVE_SIGACTION)
  struct sigaction action;
  sigaction (SIGINT, NULL, & action);
  action.sa_handler = handle_signal;
  sigaction (SIGINT, & action, NULL);
#else
  signal (SIGINT, handle_signal);
#endif /* #if defined (JITTER_HAVE_SIGACTION) */
```

`VMPREFIX_FOR_EACH_STATE`.

```
struct vmprefix_state *s;
VMPREFIX_FOR_EACH_STATE (s)
  if (s != VMPREFIX_OWN_STATE)
    printf ("This is a state different from %p: %p\n", VMPREFIX_OWN_STATE, s);
```

10.6.2 Asynchronous notifications

10.6.2.1 Notification polling and safe points

To send: `VMPREFIX_STATE_TO_SPECIAL_PURPOSE_STATE_DATA` `VMPREFIX_STATE_TO_PENDING_NOTIFICATIONS`

To receive/handle: `JITTER_PENDING_NOTIFICATIONS`

10.6.2.2 Blocking primitives

10.6.3 Signals

`struct jitter_signal_notification` `pending` `user_data`

`signal.h` `NSIG`

To send: `VMPREFIX_STATE_AND_SIGNAL_TO_PENDING_SIGNAL_NOTIFICATION(statep, signal)`

To receive/handle: `JITTER_PENDING_SIGNAL_NOTIFICATION(signal_id)` `JITTER_SIGNAL_NO`

11 Internals

11.1 Status

[*FIXME*: **Still in heavy development. At some point Jitter will be feature-complete and I expect to maintain it by only fixing bugs and porting to new platforms. Even before that point it will be reasonable to expect some degree of backward compatibility, and I may make vague promises. Not yet: everything can still change radically at any time.**]

[*FIXME*: **A couple of things could be cleaner and would benefit from a rewrite. Others are good. A few are beautiful. I should make a pass on every comment and update them: the vision of the project has widened since its inception, and the oldest comments certainly do not keep into account the current architecture. A good portion of the runtime support for VMs was written when the C code generator was still a shell script using `csplit` to isolate instruction code from one another in the input file.**]

[*FIXME*: **Documentation will be added as features mature.**]

11.2 Internals overview

[*FIXME*: **Jitter provides a friendly API usable even by motivated beginners, but its internals are *complex*. Multiple layers of CPP macros, despite the limitations of the tool. C and Gas macros used together in the same assembly source file. Code generation. M4sh. Build system hacks in order to support sub-package mode. I hardly regret any of this complexity, which is necessary and not accidental. Some factoring would be possible between library code in `jitterc` instruction representation and runtime `libjitter` instruction representation.**]

11.2.1 Program naming conventions

[*FIXME*: **The `jitter` C-code-generating program uses the namespace `jitterc` in its C source code and in the build system. This is to distinguish it from user code linked to a Jittery VM, which requires a `jitter` (no `c`) library, differently from the generator. The distinction is immaterial to the user.**]

11.2.2 Code conventions

The sources follow the GNU Coding Standards (see *GNU Coding Standards*).

[*FIXME*: **Every identifier whose name begins with an underscore character `_` is reserved for internal use. Macros with a name ending in `_` do not expand to a single C statement; the expansion may contain automatic variable declarations to be used in the following statements, or possibly as part of a `struct` definition.**]

[*FIXME*: **order of arguments: container before contained; in macros, struct names first**]

[*FIXME*: **initialize/finalize vs make/destroy**]

[*FIXME*: **long literal strings spanning multiple lines, where space permits, are separated at word boundaries, with a space at the *end* of a line.**]

[*FIXME*: **abstract data type and information hiding; performance: the critical code must be fast**]

[*FIXME*: **[When this manual will be finished: I cannot declare this in good faith now:] Every identifier not explicitly documented in this book is not intended for the user, and is subject to change in an incompatible way**]

[*FIXME*: **Booleans and `bool` / `_Bool`. Why I always write “non-false” instead of “true”.**]

[*FIXME*: **iff**]

[*FIXME*: We say "speedup" even when a variant of the code, or some alternative, or some particular mode, makes the program run *slower*: a speedup less than 1 is effectively a slowdown.]

[*FIXME*: whitespace before punctuation]

11.3 Specialization internals

11.4 Patch-ins

11.5 Rewriting internals

11.6 Dispatch models

11.7 Defective VM instructions

11.8 The Jitter build system

[*FIXME*: `configure.ac` is complex but mostly standard. The Autoconf macros intended for user configure scripts such as `AC_JITTER` and `AC_JITTER_SUBPACKAGE` are defined in `autoconf/jitter.m4`.]

[*FIXME*: The header file `jitter/jitter-config.h`, generated by at configure time, is installed. [*FIXME*: explain why.] It defines CPP macros with the prefix `JITTER_` intended not to conflict with user identifiers. Such macros are used by generated VM code to be compiled without access to the Jitter source directory, and can be used in user code as well. The Autoconf macro `jitter_define_prefixed_feature_macro`, defined in `configure.ac`, is useful to define new CPP macros with the `JITTER_` prefix from other existing macros.]

[*FIXME*: Single `Makefile.am`, in the style of Section "An Alternative Approach to Subdirectories" in *The Automake manual*.]

[*FIXME*: Most of the complexity in `Makefile.am` comes from the need of compiling the Jitter examples are compiled in non-standard way, relying on a non-installed non-subpackage Jitter. In this sense Jitter's `Makefile.am` does not provide a particularly useful example for a user simply wishing to include a Jittery VM in her own project; for this use case we recommend looking at the build systems within `example-build-systems/`, which are also considerably simpler.]

11.8.1 `switch` dispatching

[*FIXME*: slow: terrible branch target prediction, range check difficult to eliminate in practice [*FIXME*: but I think a recent GCC version has made progress: check]]

11.8.2 Direct threading

11.8.3 Minimal threading

11.8.4 No threading

11.8.4.1 Fast labels

11.9 Architecture-specific notes

[*FIXME*: I *do* have opinions about ISAs and about micro-architectures as well, but they are off-topic right now.]

[*FIXME*: **Cross-toolchains are important for support, as are good single-process emulators such as qemu-user; whole-machine emulators and physical hardware, while useful, are much less convenient. If support is lacking for an architecture, my lack of convenient access to such tools is probably the reason. If you have practical suggestions about this (not requiring non-free software), please contact me.**]

[*FIXME*: **Sixteen registers are not enough for compiled code, not to speak of Jitter. I've heard more than once the absurd argument that register renaming (at the micro-architecture level, invisible to assembly) would be a solution to this problem. Of course that is not the case: register renaming breaks some dependency chains by mapping different uses of the same architectural** *register* **to different micro-architectural registers; but if the ISA is register-starved then some data in the routine will be kept** *in memory* **rather than in architectural registers—at which point register renaming cannot help. As simple ignorance of one micro-architectural feature this mistake would be easy to forgive; but is it ignorance, or x86 rationalization?**]

[*FIXME*: **This is out of date. So as not to scare people with this warning, let it be clear that** *more* **systems are tested and supported now.**] [*FIXME*: **explain how to read the table**] [*FIXME*: **As of late 2021 very out of date: the situation is now much better than this!**]

Triplet	**Works**	**Best dispatch**
`aarch64-unknown-linux-gnu`, `qemu-user`	yes	`minimal-threading`
`alphaev4-unknown-linux-gnu`, `qemu-user`	yes	`minimal-threading`
`arm-unknown-linux-gnueabi`, BeagleBone Black	yes	`minimal-threading`
`arm-unknown-linux-gnueabi`, `qemu-user`	yes	`minimal-threading`
`avr`	no[1]	-
`avr-unknown-elf`	no[2]	-
`i586-unknown-linux-gnu`, K6 200MHz	probably[3]	`minimal-threading`
`i586-unknown-linux-gnu`, `qemu-user`	yes	`minimal-threading`
`i586-w64-mingw32`, Wine	untested[4]	?
`m68k-unknown-uclinux-uclibc`	untested[5]	?
`mips-unknown-linux-gnu`, `qemu-user`	yes	`no-threading`
`mips64-unknown-linux-gnu`, `qemu-user`	yes	`minimal-threading`
`mips64el-unknown-linux-gnu`, `qemu-user`	yes	`minimal-threading`
`mipsel-unknown-linux-gnu`, Lemote YeeLoong	yes	`no-threading`
`mipsel-unknown-linux-uclibc`, Ben NanoNote	yes	`no-threading`
`mipsel-unknown-linux-uclibc`, `qemu-user`	yes	`no-threading`
`powerpc-unknown-linux-gnu`, 750 600MHz	yes	`no-threading`
`powerpc-unknown-linux-gnu`, `qemu-user`	yes	`no-threading`
`powerpc64-unknown-linux-gnu`, `qemu-user`	yes	`minimal-threading`
`powerpc64le-unknown-linux-gnu`, `qemu-user`	yes	`minimal-threading`

[1] I should not unconditionally rely on `mmap` on small embedded systems where all the memory is executable. I should also learn how to use an AVR emulator.

[2] I enabled the newlib C library from the crosstool-ng configuration, but I see no C library at all—this might be my fault; obviously the compiler cannot build executables. In any case the Jitter runtime currently relies on `mmap`, so this configuration has no hope of working without changes.

[3] I have to replace an IDE cable and test. Old unreliable hardware.

[4] I don't feel like polluting my computer with all the dependencies for the 32-bit version of Wine. The platform is very low-priority.

[5] Testing the m68k architecture is extremely inconvenient for me: `qemu-user` is not usable for m68k, and crosstool only supports `nommu` kernels for m68k, and doesn't offer glibc as a choice; I should install a GNU/Linux distribution on a virtual machine, then use either a native compiler on the virtual machine or build a cross-toolchain by hand. It can be done, but it will be annoying.

`powerpcle-unknown-linux-gnu`, ?[6]	probably	probably `no-threading`
`riscv32-unknown-linux-gnu`	yes	`minimal-threading`
`riscv64-unknown-linux-gnu`	yes	`minimal-threading`
`s390-ibm-linux-gnu`, ?[7]	untested	?
`s390x-ibm-linux-gnu`	yes	`minimal-threading`
`sh4-unknown-linux-gnu`, `qemu-user`	yes	`direct-threading`
`sh4eb-unknown-linux-gnu`, `qemu-user`	yes	`direct-threading`
`sparc-unknown-linux-gnu`, `qemu-user`	yes	`minimal-threading`
`sparc64-unknown-linux-gnu`, `qemu-user`	yes	`minimal-threading`
`x86_64-w64-mingw32`, Wine	probably[8]	`direct-threading`
`x86_64-unknown-linux-gnu`, i7-4700MQ	yes	`no-threading`

11.9.1 Aarch64

Minimal-threading works, as long as the GCC builtin for icache invalidation is sufficient—I have no real hardware to test on and QEmu does not emulate caches.

At some point I should implement no-threading as well, since the architecture is popular.

11.9.2 Alpha

Minimal-threading works, as long as the GCC builtin for icache invalidation is sufficient.

No-threading support for Alpha should be considered a low-priority task, unless some effort emerges to resurrect the ISA in a free hardware implementation; still one of these days I might do it, just for the pleasure of working with such a pretty architecture.

11.9.3 ARM

Minimal-threading works. At some point I should implement no-threading as well, since the architecture is popular.

11.9.4 AVR

The AVR architecture is currently not supported, but the idea of running on a small embedded CPU with no operating system feels intriguing as an intellectual exercise.

I might be able to make simple VMs, probably only with simple threading models, run on mid-range models. The most powerful models addressing a 16MiB data RAM will work easily.

AVR is not supported by QEmu, but there are other free software emulators; I have to learn how to use them. I have cross compilers.

11.9.5 i386

Minimal-threading works.

At some point I should implement no-threading as well. The best route seems to be generalizing the x86_64 version with CPP conditionals.

11.9.6 m68k

Even without having ever tested a complete program I am quite confident that minimal-threading works already, as long as the GCC builtin for icache invalidation is sufficient. PC-relative addressing can be completely disabled with a GCC command-line option, and I see no other problems.

[6] QEmu seems to support little-endian PowerPC only in 64-bit mode.

[7] I can't convince `qemu-s390x` to recognize as valid the executables generated by my cross-toolchain; I guess it only supports 64-bit executables.

[8] The test suite programs currently fail cross-compiling for Wine, I guess because of a problem with the stupid file extensions. However I can generate the direct-threaded interpreter, which appears to work correctly.

Looking at the generated assembly code, and differently from what I was expecting, I see no big issue with the architecture having separate register classes for pointers and integers: GCC can still keep `union jitter_word` objects in registers, and will just generate the occasional register-to-register move instruction when a VM register allocated to the "wrong" class is being used with operations for the other.

Along with the i386/x86_64 the m68k is among the few surviving architecture I know having stack-based instructions for handling procedures, instead of branch-and-link. Branch-and-link VM operations can be supported using the same technique I am using on x86_64, with the mismatch causing the same kind of inefficiency with procedure calls. I anticipate that the problem will be more serious on m68k, its less aggressive implementations providing fewer opportunities to hide the added latency with instruction-level parallelism.

Testing the m68k architecture is extremely inconvenient for me: `qemu-user` is not usable for m68k, crosstool only supports `nommu` kernels for m68k and does not offer the GNU libc as a choice; I should install a GNU/Linux distribution on a virtual machine, then use either a native compiler on the virtual machine or build a cross-toolchain by hand. It can be done, but it will be annoying.

11.9.7 MIPS

No-threading works on 32-bit MIPS with either endianness. Minimal-threading works on 64-bit MIPS as well, again both big- and little-endian.

32-bit MIPS is very well supported and has patch-ins for branches, both unconditional and conditional, and procedures.

I made conditional branches quite efficient despite the slightly unusual ISA in this respect: MIPS has no conditional flags or separate compare instruction, and can branch conditionally on either a two-register equality/inequality comparison or a more general one-register comparison against zero. The comparison operation is part of the conditional branch instruction. To allow for comparisons not expressible within the instruction (less, less-or-equal, greater, greater-than-equal between two registers or one register and and immediate) MIPS provides a three-operand set-on-less-than instruction, setting a general register to either zero or nonzero. The other possible variants set-on-less-or-equal, set-on-greater and set-on-greater-or-equal are not provided.

Delay slots are often, even if not always, filled with nops. I can write a new pass to be run after replication to fill some delay slots with the instruction being jumped to, adjusting the target as well. This would be relatively easy, and cover other cases where the opportunity is now wasted; the same infrastructure should be reusable on SH and on SPARC, where "annulled instructions" would let me cover even more cases.

For some reason GCC generates much better code for PowerPC than for MIPS when register pressure is high, even if the two architectures are very similar. I should distill a test case and ask the GCC people.

[*FIXME*: **MIPS branches are slightly unusual, and my solution for handling them with patch-ins is nice.**]

[*FIXME*: **MIPS release 6 has some (good) new features, but I can't build a cross compiler with crosstool-ng. I should write a script to do it myself. Supporting r6-specific instructions would be nice: compact branches, particularly the two-register conditional ones, are very nice. Pseudo-direct branches are deprecated, but not yet removed, in r6. Maybe I should just use `b` and `bal` instead of `j` and `jal`, even if the range is narrower; I already use relative conditional branches. It should be possible, and easy, to generate assembly code running on both r6 and on previous versions, even if the *binary* code is not compatible; however I can't test without a cross compiler. I have no r6 physical hardware.**]

11.9.8 OpenRISC

Untested.

[*FIXME*: **I can't build a cross compiler with crosstool-ng. I should write a script to do it myself.**] As a CPU with free-hardware implementations this architecture should be high-priority.

11.9.9 PowerPC

No-threading works on 32-bit PowerPC, with either endianness. Minimal-threading works on 64-bit PowerPC as well, with either endianness.

PowerPC is well supported, and has patch-ins for unconditional branches and procedures; conditional-branch patch-ins will be added soon.

The port includes some inline assembly code to invalidate the icache, tested on the PowerPC 750 but almost certainly correct for other PowerPC models as well.

POWER is currently untested; looking at the documentation I see some differences in the required icache invalidation code; however I can conditionalize that if I ever decide to support POWER, and keep a single port for both variants.

I routinely test on an emulated 750 and occasionally on a real one, but I should try other models as well; if more recent models support PC-relative loads, which is probably the case, GCC using them might cause problems.

11.9.10 RISC-V

As a CPU with free-hardware implementations this architecture is high-priority. Minimal-threading already works.

11.9.11 s390

Minimal-threading works, as long as GCC's icache invalidation builtin is sufficient.

I routinely test the 64-bit variant s390x, which is well supported by crosstool-ng and qemu-user. At this time I can build but not run "31-bit" s390 executables; still, I do not anticipate particular problems.

11.9.12 SH

Minimal-threading is currently broken on SH, because of the architectural limitation on instruction immediate width. GCC solves the problem by using PC-relative loads of constants close to the code directly in the `text` section, a perfectly sensible strategy for ordinary C code which is unfortunately incompatible with our relocated instruction intervals. Sometimes GCC generates PC-relative loads even for accessing constant bitmasks or *offsets* to be used for loading from the stack, when the constants are too wide to fit in instruction immediates; such cases are unfortunately common in C functions containing many automatic variables, or function calls.
I need a workaround.

Of course direct threading relies on GNU C only, and therefore works reliably on SH.

As a CPU with free-hardware implementations this architecture should be high-priority. Even if not the easiest ISA to support in Jitter I find the SH an exceptionally beautiful minimalistic design, already worth supporting on esthetic merit alone.

11.9.13 SPARC

No-threading works well on both 32 and 64-bit SPARCs, as long as the GCC builtin for icache invalidation is sufficient, which it should be.

There is no support yet for conditionals, but that seems easy to add. VM procedures use the flat model and never change register windows, which is probably more efficient and certainly leaves a high number of general registers available.

[*FIXME*: **As a CPU with free-hardware implementations this architecture should be high-priority; are ASIC implementations actually available?**]

11.9.14 x86_64

No-threading works well.

x86_64 is very well supported and has patch-ins for branches, both unconditional and conditional, and procedures.

Compiling the executor with the GCC option `-mcmodel=large` is an easy and reliable way of preventing the compiler from using `%rip`-relative addressing to access data from relocated code: the trick is that x86_64 can only encode a displacement from `%rip` as a signed 32-bit field, while the large model allows for wider displacements.

[*FIXME*: **explain how the "main" procedure implementation works.**] The implementation also contains an alternative working implementation of procedures not using `call` and `ret`. It relies on using a `%rip`-relative `leaq` to load the return address into a register, and then jumping; this alternative implementation is actually a branch-and-link, but since on my machine it seems slightly slower than the other solution it is disabled. See `JITTER_BRANCH_AND_LINK_NO_CALL`.

The irritating asymmetry of the operands allowed in x86_64 comparison instructions sometimes makes the order of comparisons (for example `JITTER_BRANCH_FAST_IF_LESS_SIGNED (a, b, F)` versus `JITTER_BRANCH_FAST_IF_GREATER_SIGNED (b, a, F)`) affect performance. I can, and plan to, use `__builtin_constant_p` to rewrite the conditional when one of the operands is a known literal, but GCC gives me no way of distinguishing a memory operand from a register operand.

11.10 Porting

Jitter is trivial to port to register-based CPU architectures allowing for a sufficiently wide addressing space.

[*FIXME*: **It should be easy for accumulator- and stack-based architectures as well, as long as they can address more than 16 bits; anyway these aren't common.**]

[*FIXME*: **It is possible to have a minimal port, requiring only assembly code for a word-sized load to memory and no patch-ins; this way it's easy to test each individual feature**]

11.10.1 Required files

12 This is free software

Bibliography

[*FIXME*: **This should include comments for every entry.**]

[*FIXME*: **I should thank the GForth people here.**]

[Appel 1992] Andrew Appel. “Compiling with Continuations”, Cambridge University Press, New York, 1992.
[*FIXME*: **comment**]

[Derick+ 1982] Mitch Derick, Linda Baker. “FORTH encyclopedia — The complete FORTH Programmer’s manual”, second edition, Mountain View Press, 1982.
A reference book on figForth and the slightly later Forth 79, particularly useful to follow the details of the source code in [Ting 2013]. The remarks about dialect incompatibilities are particularly useful to read such code today: a glaring example is Forth 79’s `CREATE` behaving like figForth’s `<BUILDS`, with figForth’s `CREATE` having different semantics.

[Ertl+ 2004] M. Anton Ertl, David Gregg. “Retargeting JIT compilers by using C-compiler generated executable code”, in *Parallel Architecture and Compilation Techniques*, 2004. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.67.4231`.

[Fog 2020A] Agner Fog. “Optimizing subroutines in assembly language: An optimization guide for x86 platforms”, 2020 (updated in). Available at `https://www.agner.org/optimize`.

[Fog 2020B] Agner Fog. “The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers”, 2020 (updated in). Available at `https://www.agner.org/optimize`.

[Gudeman 1993] David Gudeman. “Representing Type Information in Dynamically Typed Languages”, technical report, Department of Computer Science, The University of Arizona, 1993. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.4394`.

[Jones+ 2012] Richard Jones, Antony Hosking, Eliot Moss. “The Garbage Collection Handbook — The art of automatic memory management”, Chapman & Hall, 2012.
This book is “the” reference on garbage collection. Its style of presentation, nonchalantly hinting at possible design alternatives and presenting algorithms in a quick way without details, while appropriate for a topic of such breadth, assumes an experienced audience. We direct garbage collection beginners to [Wilson 1992] as a first introduction; a reader having mastered Wilson’s article will thoroughly understand and enjoy this deep and subtle text.

[Rodriguez 1993] Bradford J. Rodriguez, “Moving Forth” in *The Computer Journal* issue 59, ISSN #0748-9331, January/February 1993.
This article contains a good detailed description of dispatches as used in Forth implementations, starting from indirect threading (not used in Jitter) and progressing to direct threading and other techniques such as subroutine threading, token threading and segment threading (also not used in Jitter). Dispatch in Jittery VMs is in a sense even simpler than in Forth, because Jittery VMs do not need to distinguish between “code definitions” and “colon definitions”; it is however remarkable how closely the definitions of `NEXT` in a Forth implementation match fallthrough code in Jitter: in particular Rodriguez presents “[t]he `NEXT` pseudo-code for direct threading” as

```
(IP) -> W      fetch memory pointed by IP into ‘W’ register
IP + 2 -> IP   advance ‘IP’ (assuming 2-byte addresses)
JP (W)         jump to the address in the ‘W’ register
```

Available at `http://www.bradrodriguez.com/papers/moving1.htm`.

[Saiu 2011] Luca Saiu, “Scalable BIBOP garbage collection for parallel functional programs on multi-core machines”, draft, Laboratoire d’Informatique de l’Université Paris Nord, 2011. Available at `http://ageinghacker.net/publications/gc-draft--saiu--2011.pdf`.
An ultimately unsuccessful attempt by the Author to compete with Bohem’s garbage collector playing by the same rules: mark-sweep, conservative pointer finding, no safe points.
The resulting code is not substantially faster than Bohem’s and also very complex and fragile, requiring substantial computation in signal handlers where most of the libc runtime is unusable; see [Exact pointer finding and moving], page 32.

[Shi+ 2005] Yunhe Shi, David Gregg, Andrew Beatty, M. Anton Ertl, “Virtual Machine Showdown: Stack Versus Registers”, in *VEE ’05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, 2005. Available at `https://www.usenix.org/legacy/events%2Fvee05%2Ffull_papers/p153-yunhe.pdf`.

[Shi 2007] Yunhe Shi, “Virtual machine showdown stack vs. registers”, PhD thesis, University of Dublin, 2007. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.391.1562`.

[Ting 2013]

Chen-Hanson Ting, “Systems Guide to figForth”, third edition, Offete Enterprises Inc., 2013. A very detailed description of the internal workings of a Forth system running directly on the metal, which highlights the beauty and simplicity of the Forth design. The text follows the first edition from 1981, and as was typical in the era assumes indirect threading — a technique not used in Jittery VMs. It is useful to keep the first edition as well, since some glitches in the 2013 typesetting have garbled the text between Forth words happening to resemble HTML tags, a few figures and some semantically significant markup.
The Author recommends [Derick+ 1982] as a reference to follow the details of [Ting 2013].

[Wilson 1992]

Paul R. Wilson, “Uniprocessor Garbage Collection Techniques”, submitted to *ACM Computing Surveys*, 1992. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7380`.
Even at its age this excellent introduction and survey article remains the best way for beginners to approach garbage collection implementation.
We recommend the version bearing the heading “[Submitted to ACM Computing Surveys]”; the often cited but less extensive version published in *IWMM ’92: Proceedings of the International Workshop on Memory Management* either is abridged or shows the work at an earlier stage.

Index

Appendix A Stuff still to be written

Cross-compiling: configure with `--enable-emulator="$(pwd)/libtool --mode=execute qemu-ppc -r 4.6.7 -L /home/luca/usr-cross/powerpc-unknown-linux-gnu/sysroot"`

Configuring with `--disable-shared` will speed up builds.

Why not using Gnulib in generated code or in the Jitter runtime library.

I should find a good name for this software. "`vmgen`", possibly the best choice, has already been taken by the GForth people for a project quite different from mine, but with a similar aim.

I should explicitly say that the interpreters I care about use a *linear program representation*. Tree- or graph-based interpreters are too inefficient to consider seriously. VM instruction may look like native machine instructions, but don't have to; the focus is performance, and the right solution depends on what can be implemented most efficiently in a VM.

Motivation: potentially complex instructions, not a good fit for a traditional JIT; experimenting with calling conventions.

I resisted the temptation of making the thing Lispy, mostly to get Jitter more easily acceptable to a wider community—even if I really hesitated when getting to parsing operators with precedence and associativity, and almost redid everything the sensible way, using s-expressions.

On reusing VMs.

Big and small machines. "Embededd", whatever it means.

Is it a JIT?

Not even an attempt at being safe.

Mention Forth and vmgen. Mention QEmu as well.

"Threaded-code" has nothing to do with multi-threading and concurrency.

The system is agnostic with respect to binary formats: it does not rely on ELF [*FIXME*: **not really**] and does noting unusual at link time. VMs can be made into dynamic library. All the dynamic code patching happens at runtime, after startup, on code with has been already relocated in the native architecture and endianness — some architecture-dependent part of the code patching is of course sensitive to endianness, but that can be handled with simple preprocessor conditionals (see for example MIPS instruction patching in `machine/mips/jitter/machine/jitter-machine-c.c`). All current assembly code *does* require the GNU assembler (in particular alternative macros, subsections and the section stack), and I will almost certainly rely on its features even more in the future.

The code I care about: compilers, symbolic processing. If I were doing, say, graphics, my priorities would be different and I'd have pursued some paths which I've left open.

A program using a Jitter-generated VM does not need to know the dispatch or the number of fast registers: it can be simply linked to one of several libraries generated for a given VM, all having a compatible APIs. No preprocessor feature macros are needed externally and all such libraries are functionally interchangeable.

Document how to install the Emacs major mode for Jitter, even if it is completely standard.

Every dispatch more sophisticated than `switch`-dispatching relies on GNU C, and freely uses GNU C extensions: `typeof`, computed gotos, nested functions, GCC builtins, function attributes, empty inline assembly with constraints to notify the compiler of variables changing value out of its control. More critically, some distpatching models also rely on non-trivial assumptions about code generated by GCC, which are subject to breaking with future GCC versions. A reasonable effort will be made to keep jittery code from breaking, but the user is encouraged to always rerun the Jitter test suite after upgrading GCC.

Very little assembly in the source code; almost all inline asm is actually platform-independent.

Implementation limits:

- maximum number of residuals per specialized instruction equal to the number of bits in a word. Of course this could be lifted, but it would be painful and almost certainly useless. This is because of `vmprefix_specialized_instruction_label_bitmasks` and `vmprefix_specialized_instruction_fast_label_bitmasks`.
- Depending on how I encode fast jump instruction placeholders in compiled code there will be other limit on the number of residuals, which however will be higher than the previous limit. Or maybe I can store the information away from the placeholder, and keep a pointer *to* the placeholder, keeping informations along with the pointer. Yes, this is the best solution.

`!BEGINBASICBLOCK` marks the beginning of a basic block *at the VM level.* Within each VM instruction, and from a VM instruction to the very beginning of the next (which at the VM level is fall-through control), there may be other jumps; these do not require special provisions, except making sure that each instruction is compiled into an uninterrupted interval of native instructions, without ever referring to addresses out of the interval either for native instructions or data.

A.1 (Meta-)language

Shall I replace the phrase "dispatch" with something else? I use it to describe two different runtime behaviors: *dispatching*, in the sense of jumping from one VM instruction to the next, and *argument access*. Maybe I should just say "runtime".

About the word "word": it does not mean what it means in Gas.

Replication/relocation: GForth's "dynamic superinstructions with replication" corresponds to my *minimal-threading* dispatch model; my *no-threading* dispatch is close to what `gforth-fast` did. Jitter does not support "dynamic superinstructions without replication".

GCC is *not* called at run time; that would be very slow, and the result would probably be inferior (separately compiled code could not easy share runtime state such as VM registers, without resorting to globals in memory). Gas is *not* called at run time either. In fact a Jittery VM doesn't need to invoke any external program to work except `objdump` for disassembling, and even that is only useful for debugging and developing, and therefore not critical for performance. VMs do not rely on `ld.so` for loading or relocating generated code: everything is done in memory, on raw instruction bytes. There is no `dlopen`.

Jittery VMs might be useful as a tool to experiment with garbage collectors, since it allows for low-level control; Jitter gives the user much more control over the generated code compared to a compiler, and I guess replicated code could count as a reasonable approximation of compiler-generated code when evaluating the performance of a collector. With Jitter it is easy to reserve (VM) registers for special purposes, and to play with weird stack organizations.

Polemical references to what scripting languages do, to be softened: if your program spends most of the time accessing complicated associative data structures you should use external functions for this—or even better *you should not do it*: arrays were developed for a reason. Bignums are nice to have, but should not be used for *all* numbers. Having floating-point numbers *only* is stupid for the same reason. If you have a radical idea for a computation model and you want to encode everything in it, make sure that you can recognize the optimizable cases and reduce them to the fast version almost always.

A.2 Frontend performance

`mmap`; a lot of `malloc` and `free` which are difficult to do without, because of rewriting. But custom heap allocators might be a good idea for the future, assuming the common use case is destroying a routine after it's been specialized, and keeping only the specialized version; this is currently not possible as both routine versions are kept in the same data structure, but that will change.

Even in its current state Jitter's frontend should run much faster than the alternative of spawning `gcc` or even just `as` plus `ld` processes, and then using `dlopen` to link object code.

Specialization can possibly be made faster (and the generated code certainly much smaller) by using simple automata techniques of the same family of those of Flex—the problem involves building an automaton for specialized instruction prefixes, which I think I can do directly in DFA form by using a hash at generation time. I cannot really tell how much of a problem specialization performance is at run time without benchmarking a complete Jittery application. As for compile time, specialization is heavyweight but not the bottleneck: the executor, being essentially one huge function, is more demanding on the compiler than a large set of small static functions calling each other. Switching from the current approach based on custom code generated for each specialized instruction to a single generated table accessed by hand-written code would shift the load from the instruction cache to the data cache. The table-based alternative would touch less memory, but possibly with worse spatial locality.

The frontend works in passes: instruction specialization is a separate pass, working on every unspecialized instruction one after the other. This is probably good. Will the specializer code and data remain in L1i and L1d between a specialization pass and the next? I doubt it.

A.3 Markup samples for the reference part

`SWAP` (*A*, *B*) `SWAP` *assigns the value of* `A` *to* [Macro]

B and the value of *B* to *A*.

Both *A* and *B* must be lvalues. Each is evaluated exactly once, in an unspecified order.

The macro expands to a statement, and there is no return value.

```
int a = 10, b = 20;
SWAP(a, b);
printf ("%i, %i\n", a, b);
⊣ 20, 10
```

`int sum` (*int q*, *float* `w`) [Library Function]

...

`bool frobnicate` [Global Variable]

...

A.4 GCC options

This seems reasonable: `-O2 -fno-gcse -fno-plt -fpic -fpie -fno-crossjumping -fno-reorder-blocks -fno-thread-jumps -fno-jump-tables -fno-tree-switch-conversion` even if the result is not satisfactory when fast registers are too many—except on PowerPC, where the result keeps using registers well; that is surprising as MIPS should be very similar, but the generated code is much worse.

x86_64: `-mcmodel=large` seems to avoid `%rip`-relative addressing altogether in generated code. This is possibly a breakthrough, generalizable to other architectures. Notice that my hackish way of generating assembly files hides the effect if I override CFLAGS from the `make` command line.

SH: some subset of these options seems to greatly help in generating better (floating-point) code, even if it's still polluted with `fpscr` modifications and PC-relative loads crashing my replicated code: `-Ofast -funsafe-math-optimizations -ffast-math -fpic -maccumulate-outgoing-args -mprefergot -mnomacsave -mfmovd -ftree-vectorize` I guess the `fpscr` problem would disappear if I built a toolchain with only one floating-point width.

A.5 Among design goals

Jitter makes it possible to experiment with compilation techniques: even runtime features in my opinion too inefficient to use in production should still be convenient to implement on a VM. The development of a runtime system can be made incremental, thanks to the easily modifiable nature of Jittery VMs.

A.6 About tracing JITs

They are very fashionable but I'm skeptical about them. If we are already translating all the code to an intermediate representation such as the JVM bytecode or Jitter's unspecialized code we may as well accept the small additional linear-time overhead of specialization and compile *everything* to native code. In a very dynamic environment where code is rewritten multiple times at run time (like GNU epsilon does, in some execution phases) old native code can be garbage collected with finalisers, and its space reused for new generated code.

I am personally unconvinced about the need for the additional complexity of tracing. People who disagree with me can still use Jitter for generating native code from traces, collected with mechanisms of their own.

A.7 VM instruction coarseness, or granularity

It is debatable whether VM instructions should be RISC- or CISC-like.

Native code makes the tradeoffs different from traditional threaded code, with a RISC design being more feasible, particularly elegant for a low-level source language and yielding smaller replicated code blocks: this reduced complexity should make place for increasing the number of fast VM registers, for better performance.

On the other hand for a very dynamic system a CISC approach might pay off in dynamic compilation time, reducing the number of VM instructions to work with in exchange for larger, but fewer, blocks to copy: this boils down to comparing the performance of `memcpy` against additional VM code generation and specialization, with its complicated branching and memory access patterns—and `memcpy` is the obvious winner. Larger VM instructions will also give GCC ampler opportunities to optimize generating good native instruction schedules, allocating intra-instruction temporaries in temporary registers, and possibly unrolling loops. Composing RISC-like VM instructions cannot yield native code of the same quality since GCC lacks information about the interaction between one instruction and the next, and can only work on each one in isolation.

Whether all of this makes a difference in practice remains to be seen. If you have numbers to show, I am interested.

A.8 What a "basic block" is

Each "basic block" whose entry point is marked by a `!BEGINBASICBLOCK` specialized instructions in some dispatching models is actually an *extended basic block* in the sense of a linear sequence of specialized instructions beginning with its one *entry* specialized instruction (any number of predecessors) and containing one or more *exit* specialized instructions (any number of successors, also extended basic blocks) potentially branching out of the sequence. An extended basic block may be entered only through its first specialized instruction, but specialized instructions exiting the extended basic block may occur at any point within the block, not necessarily at the end.

A caller specialized instruction, even if conditional. always ends an extended basic block since returning from the procedure will jump back to the specialized instruction immediately following it, which therefore marks the beginning of a different extended basic block. A callee specialized instruction is also always at the beginning of an extended basic block, as it can be a branch target—the difference between branch and branch-and-link being irrelevant for the purpose of basic block identification.

The definition above would allow unconditional branches, unconditional procedure returns or `!EXITVM` specialized instructions to always end a basic block, since the following specialized instruction, if any, is not reachable through straight-line code and must be a branch target, if reachable at all. Jitter does not delimit extended basic blocks in this way, instead checking (conservatively) which unspecialized instructions are possible branch *targets*. This yields slightly longer extended basic blocks, potentially containing dead code at the end. I argue that this is not a problem in practice. The alternative would be having the user annotate meta-instructions to identify which ones perform *unconditional* branches; incorrect annotations would lead to subtle bugs. Caller meta-instructions *do* require a user annotation in the form of the `caller` property, for the very purpose of correctly delimiting basic blocks, without which C macros for branch-and-link are not available to user code: but in this case the annotation cannot be forgotten if the meta-instruction code actually contains a branch-and-link operation, and even a redundant annotation does not yield incorrect code.

I'm not currently using the phrase "extended basic block" since I've seen the same phrase used for slightly different concepts. This might change in the future.

Appendix B Stuff still to be implemented

B.1 Emacs mode

Emacs mode: the empty code block here should be recognized as valid, but it isn't.

```
state-struct-c
  code
  end
end
```

B.2 `mmap`

`mmap` code to low addresses where possible. By default Linux doesn't allow user processes to `mmap` the the very first page, but using the space from the second one (usually `0x1000` or `0x2000`) to `0xffff` can be a big win for slow labels on RISCs with 16-bit-only immediates and without fast label support—however the set of those machines is supposed to become very small.

B.3 Word size on the target machine

Make sure that `jitter` doesn't expand BITS_PER_WORD and similar macros according to the machine where *it* runs; I'm almost positive it does now.

This is difficult. Shall I just generate every possibility? A heavyweight solution would be generating every possibility separately in output C code, with `#ifdef`s around to only enable the used part. This would make the generated code three times as big (even if not slower to compile) entailing three sets of definitions:

- 16-bit word — it costs nothing more at this point;
- 32-bit word;
- 64-bit word.

If I do this it is imperative to do it in a non-invasive way, possibly by having `jitter` call itself more than once, collecting and appending the generated files; this way the code change can be localized.

B.4 Non-relocatable instructions

Locally *un*wrap globals and functions. This is easy and independent from the rest.

Implement transfer registers, likely in memory, as offsets known at compile time relative to the *base*. This is easy.

Second step: generate special specialized instructions (which is to say, without unspecialized counterparts) to transfer between ordinary and transfer registers, and to load transfer registers with immediates.

Third step: do not specialize non-relocatable instructions, and insert transfer instructions around non-relocatable instructions at specialization time.

B.5 Machine register use: single *base* [almost done now]

Use a single *base* to point to memory residuals, slow registers and possibly transfer registers as well. This is how to organize the array with multiple data types:

- first come the residual memory cells, whose number and classes are known;
- then come the transfer registers held in memory, whose number and classes are again known (shall I overlay transfer registers and memory residuals? Any saving would be minimal

unless I completely eliminate one of the two concepts, which will not be feasible with multiple user-specified data types—it would be feasible now);

- then come the slow registers, ordered first by index, then by class: for example, if there are three classes *A*, *B* and *C* and fast registers are 6 (`%a0`..`%a5`, `%b0`..`%b5`, `%c0`..`%c5`) then the slow registers will be, in order, `%a6`, `%b6`, `%c6`, `%a7`, `%b7`, `%c7`, `%a8`, `%b8`, `%c8`, and so on. Slow-register residual arguments will be encoded as offsets from the base: no multiplication needed at run time. When more slow registers are needed at run time it is easy to grow the array with `realloc`: only the last part changes.

Alignment should be optimal, possibly by making each field aligned like the largest-alignment class—this is a simple solution, but it may be possible to do better.

B.5.1 Base pointer not at offset zero

The base should not necessarily point to the beginning of this memory buffer: it can point inside it, at a known offset: this would make offsets from the base smaller in modulo, giving access to the negative part as well, which might be important on architectures with only small immediates (SH, ARM).

B.6 Alias argument class

Rewriting may make some arguments of the same instruction always equal to one another; in this case only *one* of them should be residualized. This might be important with slow registers, but I haven't thought much of how to do it. It's also important to respect modes: if one alias argument has input mode and another input/output mode, I *am* allowed to modify the input/output version of it within the instruction: after all it's the same register. Hum. Maybe it's easier than I think.
It's important that this optimization is automatic: it's not acceptable to pollute a VM specification with explicit two-argument versions of meta-instructions, just as rewrite targets. Jitter should automatically discover that some arguments are redundant, and transform the definition. I guess I will need an "alias" argument class, to be used internally and possibly for the user as well (or just to make my life easier when debugging).

B.6.1 Scratch register not reserved

Do not reserve the scratch register: just mark it as clobbered or set by inline asm in residual-loading code, so that GCC can reuse it as a temporary. This would require GCC local register variables, but the thing is not trivial: residual loading snippets can no longer just be copied before each instructions, but have to become patch-ins; otherwise GCC might insert code *around* the local register variable declaration or inline asm, to save and restore the value previously held in the scratch register.

This is a lot of work. An easier compromise might be not to reserve the scratch register at all, and instead saving and restoring it from the assembly snippets setting it.

B.7 Frontend optimization

Optimize the frontend. In particular, turn checks for impossible conditions into `asserts`, and add a configure option to disable assertions. Make fast unsafe versions of instruction-appending functions, to be used from debugged C code generators – but not from the parser, whose input comes from the user. Measure performance in generated VM instructions per second or possibly native instructions per second or bytes per second; on `moore` it's currently around 1.7M VM instructions/second, including the parser overhead; I can't really tell if that's fast, without knowing the numbers for any JITs.

B.8 Frontend cleanup

Right now the frontend has data structures completely different from the ones in `jitter`. Much of this distinction is reasonable, but some encoding, particularly in instruction arguments, could be shared.

B.9 Memory directives

Let the user specify data along with instructions in symbolic VM files, and a corresponding C API.

Labels preceding instructions and data will need to be different because of the difference at specialization time, but it should not be a limitation to have them be even syntactically distinct.

B.10 Stack slot access

Slow registers mapped to a stack, to make procedure calls easier? Can I make that general? I'd say not.

But it would be easy to provide macros accessing the stack slots of a given non-TOS-optimized stack as l-values. In order to be efficient they would be provided in two versions, working with indices and offsets.

B.11 Convenience register spilling

Adding macros to spill and unspill all the fast registers, and all registers up to a given index, would be easy.

This of course would be convenient for the user but suboptimal, so the feature would consist in macros to be optionally possibly called in VM instruction code.

B.12 A ridiculous i386/x86_64 optimization

Add a different pair of call/prolog macros pushing the return address to a non-TOS-optimized stack. This would be implemented with inline assembly on x86 by an `xchg` on the `%rsp` and a VM stack pointer, and on the other side as just an xchg; and in normal C for every other architectures.

This may or may not be worth the trouble, and anyway should not be the main calling facility.

B.13 No-threading: generate one C compilation unit per VM instruction

A wild idea: instead of generating the no-threading "interpreter" as a C file containing one huge function with ~10000 labels, I could generate ~10000 small C compilation units each containing one function—one per VM instruction. In order to guarantee compatibility when composing compiled code I would have to reserve registers myself for runtime state data structures: this can be done with GCC global register variables. I would lose access to data on the stack shared by all the instructions, but I could use my base pointer instead of C's stack pointer, without any loss of efficiency.

Instead of generating ~10000 C files I could generate just one, and use CPP to disable everything but one function at each compilation. That would make things a little cleaner.

Pros:

- almost certainly better code: I get to decide exactly what part of the runtime state goes into registers.

- possibly break my dependency on GCC's `-fno-reorder-blocks`, which is very critical right now—even if nobody has talked of removing that option I'm relying on its behavior, which is not guaranteed anywhere; in fact I'm surprised that GCC can even generate assembly code exactly respecting the order in the source;
- almost certainly, avoiding implicit reliance on code and data defined in an instruction block and used by another. This currently breaks SH, where GCC uses PC-relative addressing to load constants too large to fit in immediates, which would be a good (and clever) solution normally, but a problem for my code-copying technique;
- Much less memory required for GCC.

Cons:

- How do I force GCC not to alter the stack pointer before the beginning or after the end of the code to be copied? Can I actually reserve something like `%rsp` with `-ffixed-REG`? The documentation is not completely clear, but I strongly doubt that. [In fact, I can't]. Were the QEmu people hit by this problem before they switched to their new code generator?

  I'm pessimistic here. I can keep a global register variable in `%rsp` and reliably read and write it from C even without inline asm, but I can't prevent GCC from adjusting the stack whenever it wants, possibly right before my reads or after my writes.

  Until I find a solution, this idea is on hold. Variations of this alter `%rsp` in places impossible to control:

```
//register void* rsp asm ("%rsp");
register long a asm ("%rbx");
register long b asm ("%r12");
register long c asm ("%r13");
register void* d asm ("%r14");

extern void *beginning, *end;

void
f (void *label)
{
  asm volatile ("# Function beginning");
  if (__builtin_expect (label == 0, 0))
    {
      beginning = && beginning_label;
      end = && end_label;
      return;
    }
  goto * label;
 beginning_label:
  {
    asm volatile ("# After beginning");
    a ++;

    volatile long p [3000];
    p [0] = 0;
    p [1] = 1;
    p [2] = 2;

    asm volatile ("# Before end");
  }
 end_label:
  asm volatile ("# After end");
}
```

- A lot of time required for the linker.

  I suppose that the GCC part of the compilation would still be heavyweight in time, even it it would become easy to parallelize. The stress would move from the compiler to the linker: can GNU LD efficiently link 10000 compiled modules, each containing one non-static function? That is easy to test;

- More total time required for GCC? I am curious about that;
- Nested C functions, as I mean to use them (several functions within the one interpreter function, called by replicated code from the interpreter function) would break. But maybe I can work around that;
- Possibly very disruptive in Jitter.
- What about non-word-sized elements in the status? Anyway, anything not fitting in a register would be accessible in memory base-relative. Everything would remain thread-local, which is good.